A Method for the Generation of Test Cases Based on SDL and MSCs

Jens Grabowski Dieter Hogrefe Robert Nahm

Institut für Informatik, Universität Bern
Länggassstr. 51, CH-3012 Bern

IAM-93-010

April 1993

Abstract[1]

Within this paper a method for the generation of test cases for conformance tests is presented. The method is based on a formal specification written in CCITT SDL [CCI92b] and on Message Sequence Charts (MSCs) [CCI92a]. It assumes that the purpose of a test case is given by at least one MSC. Although SDL was chosen as formal description technique and MSCs were chosen to express test purposes, in principle, the presented method should work for any formal specification which can be represented as labeled transition system and for any test purpose which can be described by a finite automaton.

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

[1]This work is performed within the F & E project, no. 233, *'Conformance Testing – A Tool for the Generation of Test Cases'*, funded by Swiss PTT.

1 Introduction

Formal description techniques (FDTs, i.e. LOTOS, Estelle, or SDL) frequently are used within industry and standardization bodies to describe the functional properties of communication systems (e.g. OSI or ISDN). FDT descriptions can be simulated and the possible interactions between a system and its environment can be generated automatically. Although test cases describe such interactions the automatic generation of test cases from FDT descriptions is still an open problem. The basic problems deal with the questions: *How long is a test case? What is the test verdict (e.g. PASS, or FAIL)?* and *What can be concluded from a test verdict?* Furthermore, there exists a gap between research and practical testing.

Research: Approaches coming from research like UIO [Wez90] or the W-method [Cho78] can handle systems with a small state space. They test every state transition exactly one time. Therefore, the length of the test cases is determined and the test verdicts are *PASS* and *FAIL*. From a *PASS* verdict a behavioural equivalence between specification and implementation can be concluded. The problems of these methods are state explosion and infinite state spaces.

State explosion occurs because of exponential relations between a specification and its state space. This means for example that the state space exponentially grows with the number of processes, or with the size of buffers. Even small examples cause problems for UIO or the W-method.

None of the mentioned methods can be applied to systems with an infinite state space. Unfortunately, FDTs force the description of systems with an infinite state space. Infinite signal queues of SDL processes or unlimited data descriptions are two examples for this. However, there can not exist test methods which guarantee behavioural equivalence for systems with an infinite state space. Even finite state machines which communicate by means of unbounded FIFO buffers (i.e. the base model of SDL) are as powerful as Turing Machines [BZ83] for which the behavioural equivalence is undecidable [HU79]. For testing the situation is more complicated since there is in general no knowledge about the whole implementation. Only the interactions between an implementation and its environment are observed for a certain time. One solution is to guarantee a finite state space by giving static restrictions to the specification. But such restrictions often are also undecidable and they do not prevent state explosion [Fin88].

Practical Testing: Real systems are very complex and approaches like UIO or the W-method can not applied. The present procedure of writing test cases is an intuitive and creative process which only is restricted by informal regulations. The intuition behind a test case is reflected by the so-called *test purpose*. A test purpose denotes an *important part of a specification* which should be tested. The meaning of the term *important part of a specification* often is a philosophical problem. Some people argue that one has to select test cases which check the normal behaviour of a system (e.g. correct data transmission), since this reflects the main purpose of a system. Other people think that one has to test the critical parts of a specification (e.g. error handling), since in general the normal cases have been tested thoroughly by the implementors.

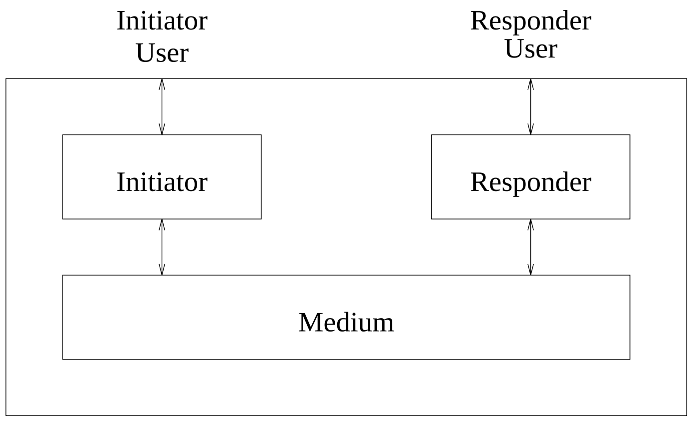


Figure 1: Architecture of the Inres protocol

Our approach: Our approach does not solve the mentioned philosophical problem but it supports practical testing. It combines test purposes defined by Message Sequence Charts (MSCs) [CCI92a, GR92] and a corresponding SDL description [CCI92b] in order to generate test cases.

MSCs (cf. Figure 2) are a widespread means for the graphical visualisation of selected system runs of communication systems [GGR]. A test purpose can be defined by an MSC in form of the required signal exchange[2]. An MSC does not define a complete test case. It does not describe the signal exchange which drives the implementation into a state from which the MSC can be performed (*preamble*). It does not define the stimuli which are necessary to drive the implementation back into an initial state after the MSC is observed (*postamble*). It does not define what to do if a signal is observed which is not defined in the MSC, and it does not describe the values of message parameters. The missing information can be provided by an additional FDT description. We choose SDL as FDT because SDL is more used within industry and standardization bodies (e.g. CCITT, ISO/IEC, or ETSI) than any other standardized FDT [Hog91a].

The rest of the paper is organized in the following way. In chapter 2 the basic ideas of our approach are presented by means of an example. We formalize our approach by relating the traces of an SDL description and an MSC. For generating the required traces SDL descriptions are interpreted as labeled transition systems and MSCs are interpreted as finite automatons. The models for this formalization are defined within chapter 3 and the test case generation is explained in chapter 4. In chapter 5 a tool is presented which implements the described method. Finally, a summary and an outlook are given.

2 The basic ideas

In the following the basic ideas for using MSCs together with SDL descriptions as the basis for test case generation are illustrated by means of an example which is taken from the behaviour of the Inres protocol [Hog91b].

[2]It should be noted that some test purposes (e.g. time constraints, or reliability requirements) can not be expressed by MSCs. But the use of MSCs for describing the class of test purposes which can be expressed seems to be common industrial practice. Therefore, we concentrate on MSCs.

2.1 Structure and behaviour of the Inres protocol

In the sequel the Inres protocol is briefly introduced. The architecture of the Inres protocol is shown in Figure 1. The Inres protocol renders a *connection-oriented service* for data transmission. It uses a *connectionless service*. Data are transported from an *Initiator* entity to a *Responder* entity. The used service is called *Medium* service. Messages exchanged between Initiator, Initiator User, Responder, Responder User and Medium are called service primitives (SPs) and the information units exchanged between Initiator and Responder are called protocol data units (PDUs).

The Inres protocol works in three phases: *connection establishment*, *data transfer* and *disconnection* (cf. Figure 2). For a connection establishment the Initiator gets a connection request CONreq from its user, then sends a CR to the Responder and waits for a connection confirmation CC in return. After receiving CC the Initiator gives a CONconf to its user and the connection is established. If a CC does not arrive within some time limit, the Initiator will retransmit CR for three times. Afterwards, the Initiator indicates the failed connection establishment by a DISind.

When the Responder receives a CR from the Initiator it gives a connection indication CONind to its user and waits for a response CONresp in return. Upon arrival of CONresp, the Responder sends a CC to the Initiator and waits for a first data package DT.

After connection establishment data can be transferred. The Initiator User gives a data request DATreq to the Initiator, which then sends a DT to the Responder and then waits for an acknowledgement AK. If the AK does not arrive within some time limit the Initiator retransmits the DT for three times. Afterwards, the Initiator assumes that the connection is distroyed and indicates this by giving a DISind to its user. If the AK arrives in time, the next data package, if present, is sent. When the Responder gets a DT form the Initiator, it acknowledges the DT with an AK and gives a data indication DATind to its user. Afterwards the Responder waits for the next DT.

A disconnection can be initiated by a DISreq from the Responder User. Upon arrival of a DISreq the Responder sends a DR to the Initiator which then indicates the disconnection by an DISind to its user.

Initiator and Responder have to use the Medium service for their communication. The Medium service can be accessed by a data request MDATreq for transmission and by a data indication MDATind for reception. The PDUs CR, CC, AK, DT and DR can be considered as being parameters of MDATreq and MDATind. The MSC in Figure 2 shows a complete system run including connection establishment, data transfer and disconnection.

2.2 Testing the retransmission of the Initiator

A suitable test architecture for testing the Initiator entity of the Inres protocol might be the distributed test method [ISO91a] as sketched in Figure 3. The architecture of the Inres protocol (cf. Figure 1) can be adjusted to the distributed test method. The Responder is replaced by the lower tester (LT) and the upper tester (UT)[3] plays the role of the Initiator User. It is assumed that the test architecture is an SDL description which can be derived from the system specification. LT and UT are modeled as SDL processes

[3]UT and LT communicate via so-called points of control and observation (PCOs) with the IUT. For simplification the PCOs are not mentioned within the test case descriptions (e.g. Figure 4 and 5), but it is assumed that each tester serves its own PCO.

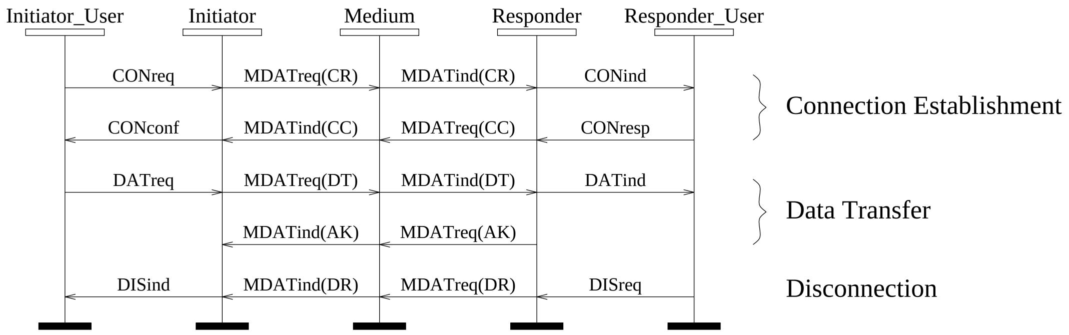


Figure 2: Complete system run of the Inres protocol

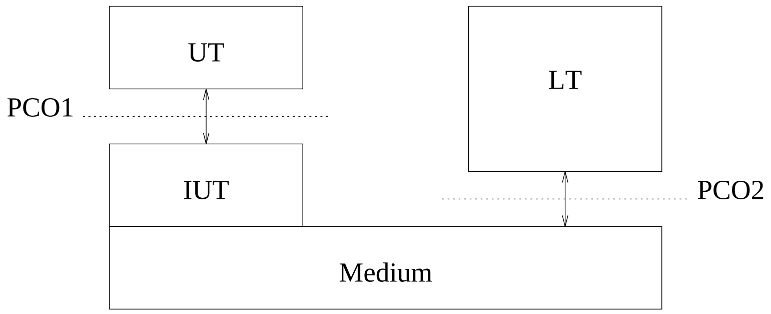


Figure 3: Distributed test method

which can send and receive any valid signal at any time[4]. A similar approach is used in [BRP89]. The system under test (SUT) consists of an Initiator implementation which is the implementation under test (IUT) and a Medium implementation which is assumed to work correct.

We want to concentrate on testing a part of the retransmission property. In particular, we want to test whether it is possible to perform a correct connection establishment after the third retransmission of the CR.

The MSC in Figure 4 shows a scenario which one may think about in the context of testing the retransmission property. The UT initiates a connection by CONreq. The LT waits for three CRs before it answers with CC which will then in return result in CONconf at the UT. Since the MSC in Figure 4 does not claim to define the entire scenario, it cannot assumed that MSCs provides complete test information.

2.3 The meaning and the representation of test cases

The method presented in this paper is based on the assumption that an MSC defines a specific part of a test case, the so-called *test purpose*. For explaining this the meaning of the terms *trace*, *observable* and *test case* has to be introduced, and the representation of test cases has to be described.

[4]For systems with a synchronous communication mechanism exists a simpler approach to define the behaviour of the tester. The inputs and outputs of the system which can be observed by its environment are inverted. Inputs become outputs and vice versa. Brinksma [Bri87] uses this technique to define the *canonical tester*.

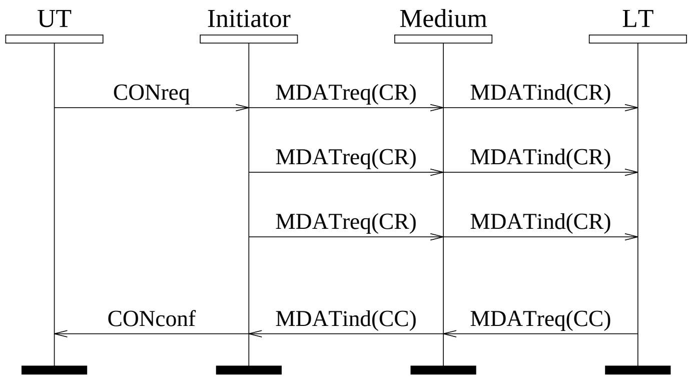


Figure 4: Connection establishment after the third retransmission of CR

Traces and observables: A *trace* describes the ordering of events which are performed during a system run. A trace of an SDL description may include the events *tasks*, *inputs*, *outputs*, *saves*, etc. of its processes. An MSC is a possible representation of an SDL trace. For testing only inputs and outputs of LT and UT are interesting[5]. Therefore, we call a trace which only includes inputs and outputs of LT and UT an *observable*.

An informal definition of test cases: A test case is defined in order to prove a specific *test purpose*. A test purpose might be a set of events which have to be performed, or a set of states which have to be reached by the IUT. A test case describes a set of observables. Each observable leads to a test verdict.

The test verdicts are *PASS*, *INCONCLUSIVE* and *FAIL*. *PASS* is given when the test purpose is reached, *FAIL* is assigned when the SUT behaves in an incorrect way and *INCONCLUSIVE* is given if neither *FAIL* nor *PASS* can be assigned.

A test case can be structured into three parts which are called *preamble*, *testbody* and *postamble*. The *testbody* describes observables which indicate that the IUT behaves according to the test purpose. The *preamble* drives the IUT from an initial state into a state from which the *testbody* can be performed. The *postamble* checks whether the testbody ends up in the correct state after it has been performed and drives the IUT back into an initial state from which the next test case can be applied.

The representation of test cases: Test cases for conformance tests are usually represented by the *Tree and Tabular Combined Notation* (TTCN) which is standardized by the ISO/IEC [ISO91b]. A TTCN test case for an Initiator implementation of the Inres protocol may look like the table in Figure 5. TTCN describes observables by means of a tree notation (cf. *Behaviour Description* in Figure 5).

The tree structure is determined by the ordering and the indent of the events. In general, the same indent denotes a branching (i.e. alternative events, e.g. lines Nr. 2 and Nr. 15 in Figure 5) and the next larger indent denotes a succeeding event (e.g. lines Nr. 1 and Nr. 2 in Figure 5).

Events are characterized by the involved instance (i.e. LT or UT), by its kind (i.e. "!" denotes an output, "?" describes an input) and by the SP which has to be send or received.

[5]It should be noted that SUT, LT and UT in general exchange SPs.

Test Case Dynamic Behaviour

Test Case Name :	Test_Case_1
Group :	Inres_Protocol/Initiator_Test/Connection_Establishment
Purpose :	Connection Establishment after the third retransmission of a Connection Request
Default :	Unexpected_Events
Comments :	

Nr.	Label	Behaviour Desription	Constraint Ref.	Verdict	Comments
1		UT!CONreq			
2		LT?MDATind(CR)			
3		LT?MDATind(CR)			
4		LT?MDATind(CR)			
5		LT!MDATreq(CC)			
6		UT?CONconf			
7		LT!MDATreq(DR)			
8		UT?DISind		(PASS)	
9		LT?MDATind(CR)		INCONC	
10		LT?MDATind(CR)		INCONC	
11		LT?MDATind(CR)		INCONC	
12		UT?DISind		INCONC	
13		UT?DISind		INCONC	
14		UT?DISind		INCONC	
15		UT?DISind		INCONC	

Detailed Comments :

Figure 5: TTCN test case for the Inres protocol

An example may clarify the notation. The statement *UT!CONreq* (cf. line Nr. 1 in Figure 5) describes the sending of CONreq to the SUT by the UT. TTCN allows to specify events with arbitrary SPs by using the OTHERWISE statement (e.g. UT?OTHERWISE in Figure 6).

Test verdicts are defined within a verdict column of the TTCN table. The verdict column of Figure 5 only includes *PASS* and *INCONCLUSIVE* verdicts. In this example *FAIL* behaviour is specified by a *default behaviour description* which is shown in Figure 6. Such defaults have to be referenced in the test case header (cf. *Default* in Figure 5).

TTCN offers much more facilities like *Constraints*, *Labels* or *Timer* which are not relevant for the understanding of this paper. A tutorial on TTCN can be found in [KW91].

Default Dynamic Behaviour

Test Step Name :	Unexpected Events
Group :	Inres_Protocol/Initiator_Test/Connection_Establishment
Objective :	Handle unexpected Signals
Comments :	

Nr.	Label	Behaviour Desription	Constraint Ref.	Verdict	Comments
1		UT?OTHERWISE		FAIL	
2		LT?OTHERWISE		FAIL	

Detailed Comments :

Figure 6: Default behaviour for the TTCN test case in Figure 5

The role of MSCs and FDT descriptions for test case generation: Up to now, the complete FDT specification of a protocol hasn't been considered but will have to be for the following reason. It is assumed that an MSC defines the *test purpose* of a test case. This means that an MSC defines a signal exchange which have to be performed by the SUT to get a *PASS*[6]. An MSC does not describe

- the *pre-* and the *postamble* of the test case,
- responses of the SUT which lead to a *FAIL* or an *INCONCLUSIVE*, and
- the parameter values of the signals which are exchanged.

In order to generate complete test cases the missing information has to be added. Therefore, an additional FDT description of the test architecture is necessary.

2.4 The observables of a test case

A test case consists of a set of observables. According to the test verdicts we distinguish between observables which lead to a *PASS*, observables which lead to an *INCONCLUSIVE* and observables which lead to a *FAIL*.

Possible pass observables: For generating a test case an observable has to be found which drives the SUT from an initial state back to an initial state, whereby the signal exchange defined within the MSC has to be performed without interrupts. We call an observable which fulfils these criteria a *possible pass observable*[7].

The observables which drive the SUT from an initial state to a state from which the MSC is applicable can be interpreted as the *preamble* of the test case and the observables which drive the SUT back into an initial state after the MSC has been applied can be interpreted as *postamble*.

We explain this by means of our test case example. The connection establishment of the Inres protocol starts in an initial state. Therefore, no preamble has to be added and our test case starts with the observable defined by the MSC in Figure 4. The MSC ends in a state where the connection is established and data can be transferred. A possible postamble is a normal disconnection which starts with the sending of MDATreq(DR) by the LT and ends with the reception of a DISind by the UT. The MSC in Figure 7 shows the MSC in Figure 4 enhanced by the disconnection. The TTCN description in Figure 5 describes the observable which is defined by the MSC in Figure 7 within the lines Nr. 1 to Nr. 8. These lines also describe the *possible pass observable* of this example. The sketched postamble is specified within the lines Nr. 7 and Nr. 8.

Inconclusive observables: If *possible pass observables* were found, observables which lead to an *INCONCLUSIVE* have to be generated. We call them *inconclusive observables*. An *inconclusive observable* has the same prefix as a *possible pass observable* but its last event is a response of the SUT which leads neither to a *PASS* nor to a *FAIL*. In our example

[6]From a theoretical point of view an MSC can be interpreted as a *liveness property* of the FDT description. It must be observable within a system run which leads from an initial state back to an initial state of the FDT description.

[7]In general there may exist more than one *possible pass observable* for a test case.

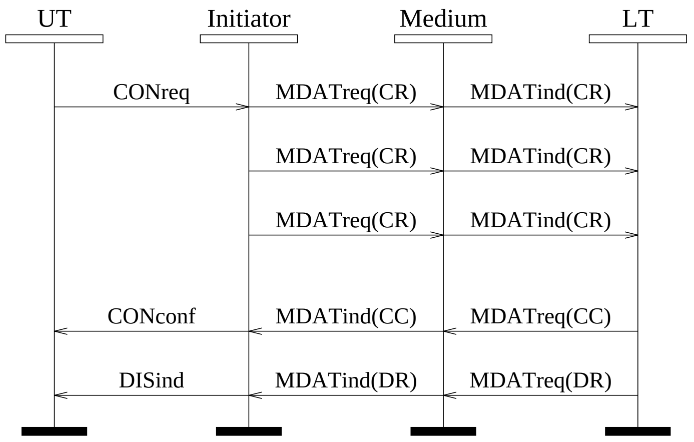


Figure 7: MSC of Figure 7 with a possible postamble

interrupts of the connection establishment by DISind lead to an *INCONCLUSIVE*. Within Figure 5 these cases are shown in the lines Nr. 9 to Nr. 15.

Fail observables: *FAIL* observables are added to the TTCN test case description by means of the *OTHERWISE* event and a default behaviour description (cf. section 2.3 and Figure 6).

Possible and unique pass observables: The *possible pass observable* of the TTCN test case in Figure 5 is shown in the lines Nr. 1 to Nr. 8. But this observable does not ensure that the MSC has been performed during a test run. After the reception of DISind a test verdict is assigned and the test case is finished. But according to the SDL description of the Inres protocol a fourth MDATind(CR) may be on the way. In this case the MSC in Figure 8 would be performed.

Such problems arise because the SUT is treated as a black box and therefore, LT and UT only have an incomplete system view. For the tester the SUT behaves in an indeterministic way. In our example the indeterminism is caused by the asynchronous communication mechanism of SDL. Without seeing all input and output events of Initiator and Medium, we can not make any assumption about an ordering, or a time relation between the DISind and a possible fourth MDATind(CR). However, if a fourth MDATind(CR) arrives, the *PASS* in Figure 5 has to be overwritten by an *INCONCLUSIVE*.

The LT does not know how long it should wait for a fourth MDATind(CR) after the reception of the DISind by the UT and before the assignment of a *PASS*. Therefore, a test run according to Figure 7 cannot be distinguished from test runs according to Figure 8 in all cases. A new postamble has to be found.

A correct postamble of our example is shown within Figure 9. Instead of MDA-Treq(CR), a data package DATreq[8] is transferred, but the reception by the LT is not acknowledged. The Initiator retransmits the data package DT three times, indicates afterwards the disconnection and goes back into a *disconnected* state. The FIFO property

[8]Data is transported as a parameter of DATreq. Since this parameter does not influence the behaviour of the Inres protocol it is omitted.

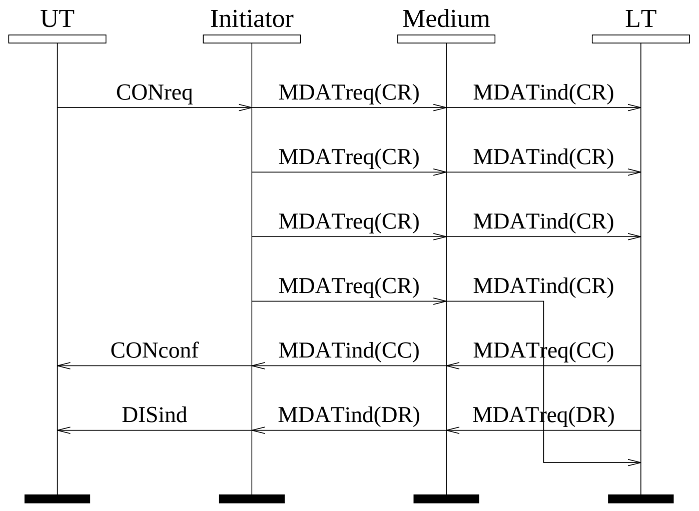


Figure 8: MSC describing a not expected system run

of queues and channels in SDL ensures that after the reception of the first MDATind(DT) no fourth MDATind(CR) can be received. Thus, the reception of the DISind allows a unique assignment of a *PASS*.

We call a *possible pass observable* which uniquely ensures that the given MSC was performed a *unique pass observable*. The complete and correct TTCN test case which ensures that the test purpose given in Figure 4 was performed is shown in Figure 10. The *unique pass observable* of this example is described in the lines Nr. 1 to Nr. 12.

3 A formal interpretation of SDL, MSCs and TTCN

In the previous section the basic ideas of the presented approach are discussed on an informal level. The central terms of the argumentation are trace and observable. In section 2 we only provide a more or less intuitive relation between the traces described by MSCs, SDL descriptions and TTCN. From an abstract point of view our method compares SDL and MSC traces and writes down certain traces with specific properties in TTCN notation.

To explain the test case generation formally we have to find a common representation for traces and observables of SDL, MSC and TTCN. Moreover, a mathematical model to generate traces for a specific SDL or MSC description is needed (see section 3.1). Furthermore, SDL, MSCs and TTCN have to be related to the chosen trace representation and to the mathematical model (section 3.2, 3.3 and 3.4). The used notation is explained in the appendix.

3.1 Trace representations and mathematical models

Communication systems are composed of several processes, which exchange signals and execute their statements independently and parallel. There are several possibilities to represent traces of such systems. They differ in the following points:

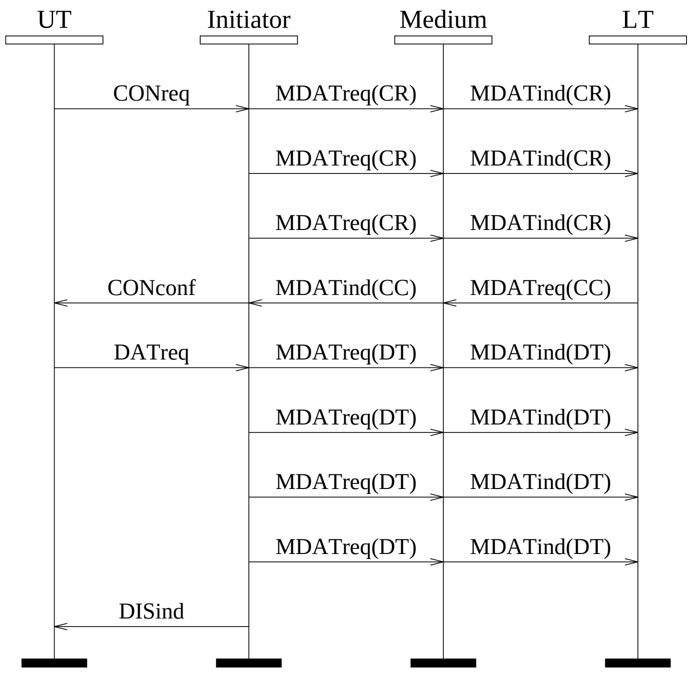


Figure 9: MSC of Figure 4 with a correct postamble

Test Case Dynamic Behaviour					
Test Case Name :	Test_Case_3				
Group :	Inres_Protocol/Initiator_Test/Connection_Establishment				
Purpose :	Connection Establishment				
Default :	Unexpected Events				
Comments :					
Nr.	Label	Behaviour Desription	Constraint Ref.	Verdict	Comments
1		UT!CONreq			
2		LT?MDATind(CR)			
3		LT?MDATind(CR)			
4		LT?MDATind(CR)			
5		LT!MDATreq(CC)			
6		UT?CONconf			
7		UT!DATreq			
8		LT?MDATind(DT)			
9		LT?MDATind(DT)			
10		LT?MDATind(DT)			
11		LT?MDATind(DT)			
12		UT?DISind		PASS	
13		LT?MDATind(CR)		INCONC	
14		LT?MDATind(CR)		INCONC	
15		UT?DISind		INCONC	
16		UT?DISind		INCONC	
17		UT?DISind		INCONC	
18		UT?DISind		INCONC	
Detailed Comments :					

Figure 10: TTCN test case description which ensures the test purpose of Figure 4

- *States* versus *events*
  A trace can be a sequence of states or a sequence of events. The relation between these approaches is, that an event is a transition between two states and a state can be interpreted as a sequence of events. For the generation of test cases only sequences of events are relevant. Therefore, in the sequel we only consider events.

- *Partial order representation* versus *interleaving representation*
  In the partial order representation, a trace is a partially ordered set of events. In the interleaving representation a trace is a sequence of events. The relation between these representations is, that a partially ordered set of events can be described by a set of sequences, where every sequence is compatible with the partial order.

- *Linear time representation* versus *branching time representation*
  In the linear time representation traces are described as a set of sequences. In the branching time representation a set of traces is described as a tree. The relation between these representations is, that every path in the tree is an element of the set.

- Different notions of *atomicity*
  Usually an event is atomic. But also certain sequences of events can be atomic. Atomicity means, that atomic sequences of events can not influence and can not be influenced from parallel executed atomic sequences of events.

Figure 11 classifies SDL, MSC, TTCN and Automata theory according to the used trace representation.

1. In SDL a trace can be defined as a sequence of events, where a sequence of events, which leads from one SDL state to the next SDL state is atomic. Furthermore, a linear time representation can be used.

2. An MSC describes a partially ordered set of events. Every event is atomic and a linear time representation is used.

3. TTCN describe traces by trees. Every node of the tree is an event and the events are atomic.

4. Automata theory [HU79] is a mathematical model, which works with sequences of atomic events. Furthermore, a linear time representation is used.

Our approach. In order to relate the traces of SDL descriptions, MSCs and TTCN we choose an automata theoretic approach and represent traces as sequences of atomic events. There are several reasons, why we choose *automata theory* as mathematical model.

- The trace representation of automata theory does not fix to the trace representation of MSCs, SDL and TTCN, but there are only a few changes necessary to adapt it. For MSCs the partial order has to be translated to a set of sequences. For SDL the atomicity has to be adapted and for TTCN the tree has to be translated into a set of sequences.

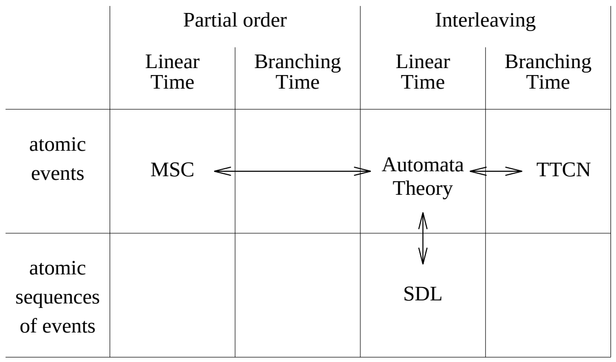


Figure 11: Classification of trace representation

- Automata theory is a well founded mathematical model, which provides a large number of useful theorems, e.g. theorems about decidability.
- Automata are easy to implement and therefore suitable for generating traces.

3.2 Specification and Description Language (SDL)

An SDL specification can be modeled by a labeled transition system, which can be interpreted as an automaton, where all states are end states. There are different methods to derive a labeled transition system from an SDL specification. One method is described in [Nah93].

Labeled transition system. A *labeled transition system* is a tuple LTS $= (Q, E, R, q_0)$, where

- Q is a set of states,
- E is a set of labels resp. events,
- $R \subseteq Q \times E \times Q$ is a transition relation and
- $q_0 \in Q$ is the initial state.

Relation between an SDL specification and a labeled transition system. Intuitively Q denotes the global system states, which are determined by the control state of the processes, the content of the signal queues and the values of the variables. The state q_0 is the initial state of the SDL specification. The transition relation R determines for every state $q \in Q$ and for every event $e \in E$ the corresponding next global state of the SDL system. The events[9] E are determined by $E = \{\tau\} \cup (\cup_{i=1}^{n}(I_i \cup O_i))$. τ is an internal event and I_i and O_i are the inputs and outputs of the i-th process. The inputs and outputs are also called *communication events*.

[9]SDL processes perform transitions from one SDL state to the following SDL state. Every transition consists of a sequence of actions e.g. input, output, task, or decision. We consider actions as events of the labeled transition system.

Observable events. Since SUT, UT and LT are specified in one SDL description and since a test case only includes the communication events of UT and LT, we define

- $OI = I_{UT} \cup I_{LT}$ to be the *observable inputs*,
- $OO = O_{UT} \cup O_{LT}$ to be the *observable outputs* and
- $OE = OI \cup OO$ to be the *observable events*.

Traces and observables of a labeled transition system. A *trace* is a sequence of events and an *observable* is a sequence of observable events. We define the traces of a labeled transition system LTS $= (Q, E, R, q_0)$ from a state in $M \subseteq Q$ to a state in $N \subseteq Q$ by:

$$Tr(\mathrm{LTS}, M, N) = \{\langle e_0, \ldots, e_n\rangle \in E^* \mid$$
$$\exists \langle s_0, \ldots, s_{n+1}\rangle \in Q^* : (s_0 \in M \ \wedge \ s_{n+1} \in N \ \wedge \ \forall i \in 0 \ldots n : (s_i, e_i, s_{i+1}) \in R)\}$$

We define the observables of a labeled transition system LTS from a state in $M \subseteq Q$ to a state in $N \subseteq Q$ by:

$$Ob(\mathrm{LTS}, M, N) = OE \copyright\, Tr(\mathrm{LTS}, M, N)$$

The traces of a labeled transition system LTS with an observable o are:

$$Tr(\mathrm{LTS}, o) = \{t \in Tr(\mathrm{LTS}, \{q_0\}, Q) \mid o = OE \copyright t\}$$

3.3 Message Sequence Chart (MSC)

An MSC describes a partially ordered set of events. It can be interpreted as a finite automaton. The automaton accepts traces, which contain the communication events of the MSC and which are compatible with its partial order.

Finite automaton. A *finite automaton* is defined by a tuple FA $= (S, E, \delta, s_0, F)$, where

- S is a finite set of states,
- E is a set of events,
- $\delta \subseteq S \times E \times S$ is a transition relation,
- $s_0 \in S$ is the initial state and
- $F \subseteq S$ is a set of final states.

Traces and observables of a finite automaton. A finite automaton FA $= (S, E, \delta, s_0, F)$ can be interpreted as a labeled transition system (S, E, δ, s_0). Thus, the traces and observables of a finite automaton are defined by the traces and observables of the corresponding labeled transition system.

Relation between an MSC and a finite automaton. The relation of the MSC and the finite automaton is explained in two steps by means of the example in Figure 12. The automaton described in the first step accepts exactly the sequences of events, which are defined by the partial order of the MSC (cf. Figure 12b). In a second step we extend the finite automaton by additional events (cf. Figure 12c).

- **Step 1:** The *Automaton 1* in Figure 12 accepts exactly the sequences of events, which are compatible with the partial order of the MSC *MSC 1*. One way for the translation of an MSC into a finite automaton is described in [GHL+92]. *MSC 1* consists of two instances `P1` and `P2`, which exchange the signal `CR` two times. It describes a partial ordered set of communication events, which allows the traces `<P1!CR,P1!CR,P2?CR,P2?CR>` and `<P1!CR,P2?CR,P1!CR,P2?CR>`. *Automaton 1* accepts these traces by transiting from the initial state s_0 to the final state f.

- **Step 2:** An MSC describes a part of the signal exchange of an SDL run by a partially ordered set of events. Our approach compares traces of a finite automaton representing an MSC and traces of a labeled transition system representing an SDL description. In order to do this, the finite automaton must also be able to accept events of the labeled transition system, which are not explicitly mentioned by the MSC.

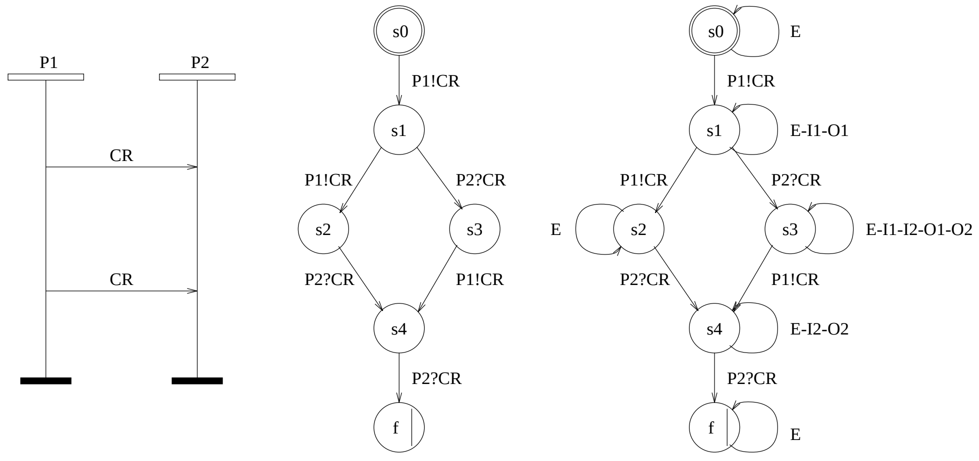
  For this aim the finite automaton is extended by Null transitions, which consume arbitrary events of the labeled transition system without changing the state. For the test case generation we require, that the signal exchange of the MSC is performed without interrupts, i.e. between two communication events on an instance axis the corresponding process is not allowed to perform further communication events[10]. To ensure this, for some states the Null transitions are restricted by certain events, which should not cause a Null transition.

  The example in Figure 12 may clarify the extension. *Automaton 2* is gained from the *Automaton 1* by introducing Null transitions for every state. Since we do not allow further communication events of an instance i between two communication events on its instance axis, we disallow its outputs O_i and its inputs I_i for some states. E.g. in state s_1 the instance `P1` has already performed the communication event $P1!CR$ and should perform the communication event $P1!CR$. Therefore, in state s_1 we exclude the outputs O_1 and inputs I_1 of the instance `P1` from the Null transitions. This fact is stated by the arrow inscription $E - I_1 - O_1$. In same manner the Null transitions of state s_2, s_3 and s_4 are constructed. In the start state s_0 and in the final state f all possible communication events E are valid. These are events of the preamble and postamble.

3.4 Tree and Tabular Combined Notation (TTCN)

A test case in TTCN describes a tree, where the nodes are observable events. We express a tree as a set of observables. The observables of a test case can be grouped into three disjoint sets - the observables, which cause a *PASS*, a *FAIL* or an *INCONCLUSIVE* verdict. We call them *pass, fail* and *inconclusive observables.*

[10]This restriction may be weakened to allow optional signals or abstractions in the MSC description.



(a) MSC 1 (b) Automaton 1 (c) Automaton 2

Figure 12: MSC and corresponding finite automata

Definition. A test case is defined by a triple $TC = (\mathit{Pass}, \mathit{Fail}, \mathit{Inco})$, where

- $\mathit{Pass} \subseteq OE^*$ are *pass observables*,
- $\mathit{Fail} \subseteq OE^*$ are *fail observables* and
- $\mathit{Inco} \subseteq OE^*$ are *inconclusive observables*

Constraints. There are restrictions on the set of observables of a test case.

- A test verdict must be unique. There is no observable, which cause two verdicts at once. Formally, this is expressed by:

  $$\mathit{Pass}, \mathit{Fail}, \mathit{Inco} \text{ are pairwise disjoint}$$

- After deriving a test verdict, it is assumed, that the test case is finished and could not be continued. This can be expressed by:

  $$\forall v, w \in \mathit{Pass} \cup \mathit{Fail} \cup \mathit{Inco} : v \not\sqsubseteq w$$

- The tree of a test case can not have arbitrary branching. UT and LT perform a sequence of fixed outputs, and afterwards they have to wait for the reaction of the SUT. Then UT and LT again can perform a sequence of fixed outputs. This means for outputs a test case has a branching of size 1 and for inputs a test case can have arbitrary branching. We express that by the notion of *alternative* observables. Formally, this is expressed by

  $$\forall v, w \in \mathit{Pass} \cup \mathit{Fail} \cup \mathit{Inco} : v \ \mathit{altto}\ w$$

We introduce two notations of alternative observables.

- An observable v is *alternative* to an observable w, if they have a prefix p in common and the first elements, in which they differ is an input. Formally, this is denoted by v *altto* w *iff*

$$\exists p, v', w' \in OE^*; a, b \in OI: v = p \cdot a \cdot v' \wedge w = p \cdot b \cdot w' \wedge a \neq b$$

- An observable v is *minimal alternative* to an observable w, if v is an alternative to w and v is only one element longer than the common prefix. Formally, this is denoted by v $\underline{altto}$ w *iff*

$$\exists p, w' \in OE^*; a, b \in OI: v = p \cdot a \wedge w = p \cdot b \cdot w' \wedge a \neq b$$

Test verdicts for a test case. An implementation I is driven according to the test case $TC = (Pass, Fail, Inco)$ and performs the observable w. Whether an observable proves the test purpose is defined by the test verdict $verdict(w, TC)$. We give a *PASS*, if a prefix of w is a *pass observable*, we give a *FAIL*, if a prefix of w is a *fail observable* and we give an *INCONCLUSIVE*, if a prefix of w is an *inconclusive observable* or the implementation does not respond during test time and w is a prefix of a *pass observable*. Formally the test verdict is defined by $verdict(w, TC) =$

- $PASS$ *iff* $\exists v \in Pass : v \sqsubseteq w$
- $FAIL$ *iff* $\exists v \in Fail : v \sqsubseteq w$
- $INCONCLUSIVE$ *iff* $(\exists v \in Inco : v \sqsubseteq w) \vee (\exists v \in Pass : w \sqsubset v)$

4 Formalizing the test case generation

Now we know that a TTCN test case consists of three disjoint sets of observables and each set corresponds to a test verdict. In this section we define the relation between observables, representing a test case, a labeled transition system representing an SDL description and a finite automaton representing an MSC.

4.1 Informal relation between SDL descriptions, MSCs and test cases

An implementation gets a PASS verdict, if we can assume, that it performs a trace from its initial state to its initial state and executes the communication events of the MSC. It gets an INCONCLUSIVE verdict, if the performed observable is specified by the SDL description but the communication events of the MSC are not performed or the initial state is not reached again. Finally, it gets a FAIL verdict, if it presents an observable, which is not specified by the SDL description. This means, that we are calculating three sets of observables - the *pass*, *fail* and *inconclusive observables.*

- A *pass observable* is an observable, from which we can conclude, that the labeled transition system representing the SDL description performs a cycle from its initial state back to the initial state and the finite automaton representing the MSC transits from the initial state to a final state.

- An *inconclusive observable* is an observable of the labeled transition system, which has a prefix with a *pass observable* in common, but the first event, in which they differ is an input.
- A *fail observable* is not an observable of the labeled transition system. It is an arbitrary sequence of observable events, which has a prefix with a *pass observable* in common. The first event, in which they differ, is an input.

For defining the three sets of observables formally, we need the notion of *possible* and *unique pass observables.*

4.2 Possible and unique pass observables

A *possible pass observable* is an observable, where the labeled transition system LTS can perform a cycle from the initial state to the initial state and the finite automaton FA transits form the initial state to a final state. We define the set of *possible pass observables* *PPO* by :

$$PPO = Ob(\text{LTS}, \{q_0\}, \{q_0\}) \cap Ob(\text{FA}, \{s_0\}, F)$$

A *possible pass observable* does not ensure, that every corresponding trace leads the labeled transition system from its initial state back to its initial state and the finite automaton transits from its initial state to a final state. For this aim we define so called *unique pass observables* *UPO* by:

$$UPO = \{w \in PPO \mid \overline{Tr(\text{LTS}, w)} \subseteq [Tr(\text{LTS}, \{q_0\}, \{q_0\}) \cap Tr(\text{FA}, \{s_0\}, F)]\}$$

Since we only consider the maximal corresponding traces of an observable w: $\overline{Tr(\text{LTS}, w)}$, this definition works only if the initial state of the labeled transition system is a stable state, i.e. only observable events can cause progress in the initial state.

4.3 Pass, Fail and Inconclusive observables

Now we define a test case $TC = (Pass, Fail, Inco)$ for a labeled transition system LTS and a finite automaton FA .

- ***Pass.*** For the *pass observables* of the test case *Pass* we take a subset of the shortest *unique pass observables* $\underline{UPO}$ (see 1.). Each element of the *pass observables* must be alternative to each other element (see 2.) and there is no further shortest *unique pass observable*, which is not alternative to all *pass observables* (see 3.).

  1. $Pass \subseteq \underline{UPO} \wedge$
  2. $\forall v, w \in Pass : (v \neq w \rightarrow v \; altto \; w) \wedge$
  3. $\forall v \in \underline{UPO} : (v \in Pass \vee \exists w \in Pass : \neg(v \; altto \; w))$

- ***Inco.*** For the *pass observables Pass* we define the shortest *inconclusive observables* of the test case *Inco. Inco* denotes the minimal alternative observables of *Pass.*

$$Inco = \{v \in Ob(\text{LTS}, \{q_0\}, Q) \mid \exists w \in Pass : v \; \underline{altto} \; w\} - pref(Pass)$$

- ***Fail.*** The *fail observables Fail* are the minimal alternatives of the *pass* and *inconclusive observables.*

$$Fail = \{v \in OE^* \mid \exists w \in Pass : v \; \underline{altto} \; w\} - pref(Pass \cup Inco)$$

4.4 Calculation of a test case

Within the previous section a test case for a given labeled transition system and a given finite automaton is defined, but there is no algorithm to calculate it. By calculating a test case we have to solve a typical reachability problem, i.e. sometimes a certain event is executed or a certain state is reached.

For finite automata the reachability problem is solved and there exist efficient algorithms to calculate shortest traces, which lead to a certain state or contain a certain event [HU79]. But the decidability of the reachability problem of the labeled transition system depends heavily on its design. In [BZ83] it is proved, that the reachability problem for communicating finite state machines, which communicate by means of unbounded FIFO buffers is undecidable. Subsequently the reachability problem for a labeled transition system, which represent asynchronously communicating processes e.g. like SDL descriptions, is undecidable.

One way to search for observables with required properties is to simulate the labeled transition system and the finite automaton in parallel. There are different search methods, like depth search and breadth search. Breadth search is not usable, since it is impossible to store all states[11]. Also depth search is not applicable, since it is not possible to guarantee termination.

Therefore, we use a k-bounded depth search. A k-bounded depth search evaluates all possible traces of length k. If no trace with required properties is found, then the search may be repeated with a higher bound or stopped the search without results.

The procedure of generating test cases. The procedure for generating a test case based on an SDL description and an MSC can be structured in four steps:

- **Step 1:** In a k-bounded depth search with increasing bound k *possible pass observables* are calculated.
- **Step 2:** Based on the *possible pass observables* we calculate the *unique pass observables*. If there are no *unique pass observables* we go back to step 1.
- **Step 3:** We choose a subset of the shortest *unique pass observables*, which are alternative to each other. This are the *pass observables* of our test case. Based on the *pass observables* we calculate the corresponding *inconclusive observables*.
- **Step 4:** The *pass observables* and the *inconclusive observables* have to be transformed into TTCN. Furthermore the *fail observables* have to be added by means of the default behaviour.

Open problems. In this paper the problem of generating test cases from SDL descriptions and MSCs is transformed into a search problem. The expense of the search heavily depends on the SDL description, which represents the test architecture (cf. section 2.2). The test architecture is derived from a system specification by omitting parts, which are not tested and by adding SDL processes for LT and UT. LT and UT are modelled as processes, which can receive and send any valid signal at any time.

[11]Note a state represents a global state of the SDL system, e.g. the control states, contents of the queues and the values of the variables.

An open problem is the optimal modelling of the test architecture, especially of LT and UT, since this may decrease the search expense. The optimization of LT and UT may include restrictions on the sequences of signals, which can be sent and received, and restrictions on the values of signal parameters.

5 On the implementation of the presented method

The presented method for generating test cases is developed and implemented at the University of Berne within the research project[12] *'Conformance Testing - A Tool for the Generation of Test Cases'*.

Figure 13 presents the architecture of the implemtation. The test case generation tool is structured in the three parts *SDL simulator*, *MSC simulator* and *test case generator*. Both simulators consist of a *transformator* and an *interpreter*. The transformators read descriptions in phrase representation of SDL (SDL/PR) and MSC (MSC/PR) and transform them into internal representations. Afterwards the internal representations are simulated by the interpreters. The test case generator is structured in four modules:

- Calculation of *possible Pass observables.*
- Calculation of *unique Pass observables.*
- Calculation of *Inconclusive observables.*
- Generation of the corresponding TTCN/MP[13] code.

The tool is implemented on Sun workstations. Its inputs are MSC/PR and SDL/PR descriptions [CCI92a, CCI92b], and its output is a TTCN/MP description [ISO91b]. Front- and backends of the tool are commercial SDL, MSC and TTCN editors.

6 Summary and outlook

A method for the generation of test cases based on SDL descriptions and MSCs is presented. The approach assumes that the purpose of a test case is given by at least one MSC. Furthermore, the problem of assigning unique test verdicts is discussed and a solution by defining *unique pass observables* is presented. The whole approach is formalized by relating the traces of a labeled transition system representing an SDL description to the traces of a finite automaton representing an MSC. The method is implemented and its applicability for real systems will be proven within a following case study. Although this paper presents the approach by means of SDL and MSCs, it may be possible to apply it to any specification which can be represented by a labeled transition system and to any test purpose which can be represented by a finite automaton.

[12]F & E project, no. 233, funded by Swiss PTT.

[13]TTCN/MP denotes the machine processable form of TTCN.

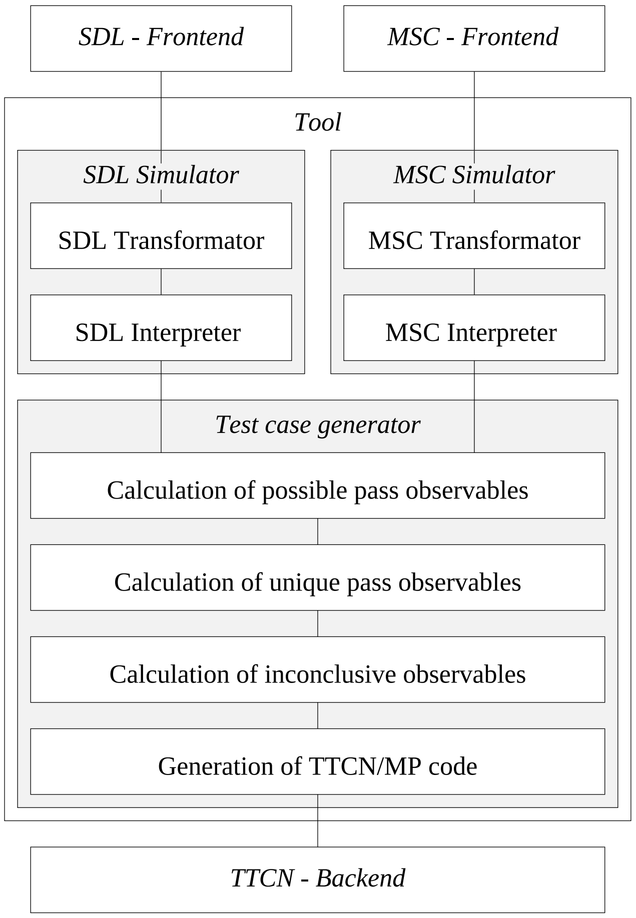


Figure 13: The tool architecture

References

[Bri87] E. Brinksma. On the existence of canonical tests. Technical Report INF-87-5, University of Twente, Niederlande, 1987.

[BRP89] Anne Bourget-Rouger and Combes Pierre. Exhaustive validation and test generation in Elvis. In O. Faergemand and M.M. Marques, editors, *SDL'89: The language at work*, volume 4 of *Proceedings of the fourth SDL Forum*, pages 231–245. North Holland, 1989.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite state machines. *Journal of the Association for Computing Machinery*, 30(2):323–342, April 1983.

[CCI92a] CCITT SG X. Message Sequence Chart (MSC). Recommendation Z.120, 1992. Geneva.

[CCI92b] CCITT SG X. Specification and description language (SDL). Recommendation z.100, 1992. Geneva.

[Cho78] T.S. Chow. Testing software design modeled by finite state machines. *IEEE-SE*, 4(3):178–187, 1978.

[Fin88] Alan Finkel. A new class of analyzable CFSMs with unbounded FIFO channels. In S. Aggrawal and K. Sabnani, editors, *Protocol Specification, Verification and Testing*, volume 9 of *Proceedings of the IFIP WG 6.1 eighth International Symposium on Protocol Specification, Testing and Verification*, pages 283–294. North Holland, 1988.

[GGR] Jens Grabowski, Peter Graubmann, and Ekkart Rudolph. The standardisation of Message Sequence Charts. submitted to the Software Engineering Standards Symposium '93.

[GHL$^+$92] Jens Grabowski, Dieter Hogrefe, Peter Ladkin, Stefan Leue, and Robert Nahm. Conformance testing - a tool for the generation of test cases. Interim Report of the F & E project contract no. 233, funded by Swiss PTT, 1992.

[GR92] J Grabowski and E Rudolph. Message Sequence Charts (MSC) - a survey of the new CCITT language for the description of traces within communicating systems. Technical Report IAM-92-022, University of Berne, November 1992.

[Hog91a] Dieter Hogrefe. Conformance testing of communication protocols in the framework of formal description techniques. Technical Report IAM-91-007, University of Berne, Institut for Informatik, 1991. 32 p.

[Hog91b] Dieter Hogrefe. OSI formal specification case study: The INRES protocol and service. Technical Report IAM-91-012, University of Berne, 1991.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[ISO91a] ISO/IEC JTC 1/SC 21 N. Information technology - Open System Interconnection - conformance testing methodology and framework - Part1-5. International Standard 9646, ISO/IEC, 1991.

[ISO91b] 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, 1991.

[KW91] Jan Kroon and Antony Wiles. A tutorial on TTCN. In *Protocol, Specification, Testing and Verification*, volume 11 of *11th International IFIP WG6.1 Symposium on Protocol, Specification, Testing and Verification*, pages 40–92, 1991.

[Nah93] Robert Nahm. Semantics of simple SDL. In *GI/ITG Fachgespräch*, 1993.

[Wez90] Clazien D. Wezeman. Protocol conformance testing using multiple UIO-sequences. In E. Brinksma, G. Scollo, and C.A. Vissors, editors, *Workhop on Protocol Specification, Verification and Testing*, volume 9 of *Proceedings of the IFIP WG 6.1 ninth International Symposium on Protocol Specification, Testing and Verification*, pages 131–143. North Holland, 1990.

A Sequences

Set of sequences. Let A be an arbitrary set, then we define the following three sets

- A^* are the finite sequences over A,
- A^ω are the infinite sequences over A and
- $A^\infty = A^* \cup A^\omega$ are the finite and infinite sequences.

Operations on sequences. Let $S \subseteq A^\infty$, $t, u, v \in A^\infty$ and $a, b, c, d, a_0, \ldots, a_n \in A$

- $\langle\rangle$ is the empty sequence,
- $\langle a_0, \ldots, a_n \rangle$ is the finite sequence consisting of the elements $a_0, \ldots, a_n$,
- $t \cdot u$ denotes the concatenation of t and u (Note, if t is infinite the $t \cdot u = t$),
- $t \sqsubset u$ "t is a strict prefix of u" holds, iff $\exists v \neq \langle\rangle : t \cdot v = u$,
- $t \sqsubseteq u$ "t is a prefix of u" holds, iff $\exists v : t \cdot v = u$,
- $\#t$ denotes the length of t (Note, if t is infinite then $\#t = \infty$),
- $a \copyright t$ denotes the filtered trace of t, which contains only the element a,
  e.g. $a \copyright < a, b, a, c >=< a, a >$. As a generalisation of this *filter* operation, the first operand may also be a set,
- $f : A \rightarrow A'$ can be canonically extended to sequences, by
  $f(< a_0, \ldots, a_n, \ldots >) =< f(a_0), \ldots, f(a_n), \ldots >$,
- $f : A^\infty \rightarrow A'^\infty$ can be canonically extended to sets of sequences by
  $f(S) = \{f(t) \mid t \in S\}$,
- $\mathit{pref}(S) = \{t \mid \exists u \in S : t \sqsubseteq u\}$, is the set of prefixes of S,
- $\overline{S} = \{t \in S \mid \neg\exists u \in S : t \sqsubset u\}$ are the maximal sequences and
- $\underline{S} = \{t \in S \mid \forall u \in S : \#t \leq \#u\}$ are the shortest sequences.

B Abbreviations

CCITT	Comité Consultativ International Télégrahique et Téléphonique
ETSI	European Telecommunications Standards Institute
FDT	Formal Description Techniques
ISO/IEC	International Organisation for Standardisation/International Electronical Commission
IUT	Implementation Under Test
LT	Lower Tester
MSC	Message Sequence Chart
MSC/PR	MSC Phrase Representation
OSI	Open Systems Interconnection
PCO	Points of Control and Observation
PDU	Protocol Data Unit
SDL	Specification and Description Language
SDL/PR	SDL Phrase Representation
SP	Service Primitive
SUT	System Under Test
TTCN	Tree and Tabular Combined Notation
TTCN/MP	Machine Processable TTCN
UIO	Unique Input Output sequences
UT	Upper Tester