Formal Aspects of Computing (1998) 10: 483–508
© 1998 BCS

Formal Aspects of Computing

Case Study in Protocol Validation: Validating an ATM Signalling Protocol

Theofanis Vassiliou-Gioles and Ina Schieferdecker

GMD FOKUS, Berlin, Germany

Keywords: ATM signalling; SDL; Validation

Abstract. This paper discusses validation techniques for communication protocols and analyzes the practical use of selected validation techniques in an automated manner. A case study on validating the ATM (Asynchronous Transfer Mode) Signalling Protocol as specified by ITU-T in Q.2931 is used for this analysis. Different error classes are identified and validated. An assessment of the different validation techniques in terms of effort and quality of results is given. As a result of the case study, an evaluation of the Q.2931 SDL specification completes the paper.

1. Motivation

This paper uses the notion validation to denote the process of finding errors of typical error classes in a communication protocol as opposed to the notion of verification, which denotes the process of proving that typical errors of a considered error class are absent. Both, validation and verification are used to demonstrate that a protocol can provide typical functions or services. This paper deals with the state based validation techniques of communication protocols [Hol91]. A communication protocol is modelled by a set of communicating extended finite state machines (EFSM). Different error classes for such protocol specifications can be identified and divided into the following groups:

- **Model error** classes are error classes, which can be derived from the underlying model of the specification.
- **Protocol error** classes are error classes, which can be derived from the concrete protocol and its requirements.

Correspondence and offprint requests to: T. Vassiliou-Gioles, I. Schieferdecker, GMD FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany. E-mail: {Vassiliou, Schieferdecker}@fokus.gmd.de

The model error classes for communicating extended finite state machines (EFSM) contain errors (e.g. [Hol91], [Wes92]) like:

1. **Deadlock:** an automaton has a deadlock if it is in a state having no transition bringing it into a subsequent state.
2. **Livelock:** an automaton has a livelock, if it cycles through a set of states, without real progress.
3. **Reception error:** a reception of a signal in a state with no defined transition for this signal is called a reception error. In this case, the protocol is underspecified.
4. **Unreachable protocol parts:** a protocol for not being overspecified should not have unreachable parts.

Errors in the protocol error class cover:

5. **Loss of synchronization:** if forbidden state combinations of protocol entities and/or processes occur, this is called loss of synchronization.
6. **Inaccessibility of reasonable states:** a protocol must reach all required and reasonable state combinations of its protocol entities and/or processes.
7. **Injury of required execution sequences:** a protocol must be able to perform all required execution sequences[1].

The main intent of this case study was not only an exercise in protocol validation, but rather the development of a validated protocol specification that can be used as a basis for test suite development for ATM signalling. In particular, we used the validated protocol specfication for cross-simulation with a TTCN test suite to obtain a more correct test suite, which is consistent with the protocol specification. Such an approach is of particular use for test suites developed independently from the SDL protocol specification and bear therefore the risk of errors, ambiguities and inconsistencies. The concrete cross-simulation has been executed against the validated and corrected ITU-T SDL specification and the ATM Forum ATM signalling test suite. A number of errors in the test suite were detected. A first report on this type of test suite validation was given to the ATM Forum test working group [ScR96].

This paper is structured as follows: In Section 2, validation techniques based on reachability analysis are discussed. Section 3 presents the SDL specification for ATM signalling and explains specification decisions. After presenting both, the validation technique used and the protocol to be validated, automated validation by use of an SDL validation tool called SDT is presented in Section 4. The parameterization of the validator, its execution and a general overview of the results of the model-specific error classes, which were detected in the protocol, are discussed in Section 5. Results are analysed in more detail in Section 6. Sections 7 and 8 present specific validation methods based on MSCs and user rules and give the respective results. The paper is summarized and concluded in Section 9.

2. Validation Based on Reachability Analysis

The main idea of any state based validation method is the exploration of the protocol state space. Different techniques have been developed, e.g. [Wes92] considers

[1] Such execution sequences might be represented by MSCs ([Ek93, ITU96]).

- Duologue matrix analysis and
- Reachability analysis.

The **duologue matrix analysis** [Zaf78] was one of the first techniques in the field of automated validation. Because of the inefficiency [Wes92] and the limitation to pairs of communicating finite state machines (FSMs), the usability of this method is very poor and therefore not considered in the following.

Reachability analysis methods try to reach every possible state starting from an initial state and analyze them. A state graph containing all reachable states and execution sequences is generated recursively by applying depth-first-search or breadth-first-search algorithms.

Reachability analysis methods are distinguished by their algorithms of traversing the state graph[2]. According to [Hol91], random walk, full search, and controlled partial search are the most useful algorithms for a reachability analysis.

The states in the state graph will be examined for the occurrence of errors like deadlock or the lack of defined transition for an incoming signal (error classes 1 and 3). In addition, the generated state graph can be examined if it contains livelocks, loss of synchronization or unreachable protocol parts (error classes 2, 4 and 5).

2.1. Full State Space Search

The full state space search algorithm examines all reachable system states. A system state consists of the control states per EFSM of the system, their waiting queues and their variables.

A full space search algorithm detects, which system states are reachable and which are not. Every reachable system state and the execution sequences of the systems can be analyzed whether a deadlock, a livelock or a reception error occurred. After finishing the complete state space exploration, unreachable parts of the protocol can be identified. A properly defined protocol does not have unreachable control states. [Hol91] stresses the fact that the amount of unreachable system states is significantly higher than the amount of reachable ones.

The major problem of the full state space analysis is its restricted applicability to real communication protocols. Even the analysis of small and simple protocols exhausts the available memory. For this reason, only protocols having up to 10^5 system states can be analyzed. Simple protocols with only some variables and very limited queues already reach this system state space size [Hol91]. This is a well known problem and often referred to as the state space explosion problem.

Full state space exploration techniques show good validation results and can to check the correctness of 'small' protocols [Wes92]. In the case of 'large' protocols, the full state space exploration techniques result only in a partially explored system state space. Most importantly, if the state space exploration is stopped when available memory is exhausted, the system state search is interrupted at uncontrolled and arbitrary points yielding to partial and hard to assess state space

[2] It should also be noted that state graphs can either be built completely before any state graph traversal is executed, or on-the-fly where states and execution sequences are validated until certain termination conditions hold (e.g. a deadlock is found). However, the complete versus on-the-fly generation is mainly an issue of main memory, disk space, and time.

coverages. Hence, the full search is reduced to a partial one without having a guarantee that the important parts of the system state space have been examined.

2.2. Controlled Partial Search

To overcome the shortcomings of full state space exploration techniques for 'large' protocols, special techniques for a systematic and controlled partial search were developed. These techniques assume that the amount A of system states that can be analyzed is significantly smaller than the amount R of reachable system states. Partial search techniques try to examine those system states, which

1. guarantee that important protocol functionality are examined and
2. that the search quality, i.e. the possibility to find an error, is better than A/R[3].

[Hol91] considers the following techniques for partial search:

- Depth-Bounds
- Scatter Search
- Guided Search
- Partial Orders
- Random Selection

The first four techniques try to forecast, in which part of the protocol errors are potentially located. For example, the scatter search tries to provoke a deadlock by favoring the reception of signals instead of sending signals in order to reach a system state, where all waiting queues are empty (which would be a deadlock situation in the protocol). However, it should be noted that such search techniques are not necessarily successful. Since the system state space is explored according to predefined search algorithms, some system states are strictly excluded from being analyzed[4].

2.3. Random Selection

In contrast to the scatter search techniques, random selection techniques have the ability to explore all system states (under the assumption that execution paths, which are complete or have sufficient depth, can be analyzed within the available memory) and are therefore potentially capable of identifying all protocol errors.

In a random selection analysis, or random walk, the next system state, which has to be explored, is chosen randomly. The random walk is executed until an error is detected, an already examined system state is reached or a sufficient depth, which has to be defined in advance, is reached.

[Wes92] shows that the efficiency of random selection techniques is identical for simple and complex protocols. Efficiency is the ratio between errors being detected to the total amount of errors existing in a protocol. Even unbounded systems can be analyzed with this random selection techniques. It is also pointed

[3] For a first approach it can be assumed that if only a fraction A/R of the protocol is examined only the fraction A/R of errors can be found.

[4] And according to Murphy's law, these system states that are excluded from the exploration, are the ones that bear protocol errors.

out that similar protocol errors occur in many system states. For fixing such errors, it is sufficient to find one of them. These are the reasons the paper uses only random selection techniques.

However, the random walk technique is not applicable if one has to identify unreachable parts of the protocols. It cannot be concluded that an unexplored protocol part is not reachable, since there is no criteria to assess whether all reachable system states have been explored. Therefore, the error class "unreachable protocol parts" cannot be validated with a random walk analysis. Another problem is to determine the depth at which it is sufficient to abort the state exploration. On the one hand, a small search depth bears the problem of overseeing many protocol errors. On the other hand if the search depth is too high, possibly a number of already explored system states are analyzed several times without finding any new protocol errors.

3. The SDL Specification for ATM Signalling

An SDL specification [ITU93] describes the functional behaviour of a system in terms of stimuli and reactions at its interfaces and defines the sequence of reactions for every possible stimulus. SDL is based on the following concepts:

- A system consists of the elements system, blocks, channels, processes, services, signal routes and signals.
- The behaviour is described by means of communicating EFSMs, which are called processes.
- Processes communicate asynchronously via channels. Each process has an unbounded input queue.
- Data are specified with abstract data types.

The ATM Signalling Protocol in [ITU95] specifies the procedures and rules for the establishment, maintenance and release of network connections at the B-ISDN user-network interface (UNI). The procedures are defined through message exchange. It distinguishes between a user and a network side. Both sides are defined as SDL blocks, built from four processes. Processes with similar names like Co_Ord_N and Co_Ord_U have similar behaviours. Figure 1 shows a simplified block diagram for the user side as an example for a peer entity.

3.1. Process Co_Ord_X

The coordination process Co_Ord_X[5] manages the distribution of incoming signals to the individual processes. It is instantiated once at system instantiation time. The process distinguishes between:

1. primitives exchanged between the processes Q_2931_X, Reset_Start_X, Reset_Response_X or the SAAL block and the application X and
2. messages that are exchanged between the above mentioned processes.

[5] The identifier x will always be used, when processes are referenced that exist at the user- and at the network side.

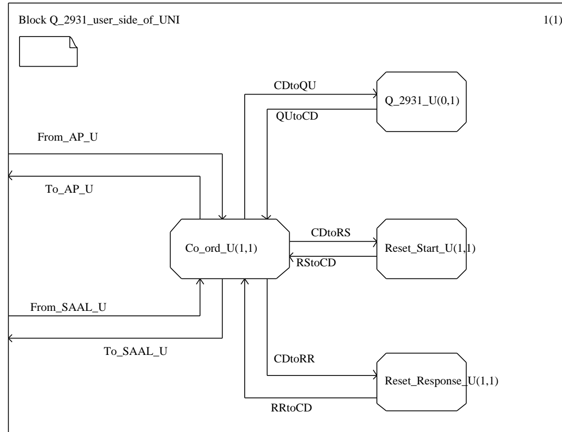

Fig. 1. Q.2931 SDL block diagram.

Table 1. State Overview for Q_2931_X.

Name	Network Side	User Side
null	N0	U0
call initiated	N1	U1
outgoing call proceeding	N3	U3
call delivered	N4	U4
call present	N6	U6
call received	N7	U7
connect request	N8	U8
incoming call proceeding	N9	U9
active	N10	U10
release request	N11	U11
release indication	N12	U12

Primitives are signals of local significance and are not transmitted to the peer entity. Messages are signals, which are transmitted to the peer-entity with the help of the Signalling ATM Adaptation Layer (SAAL). In the process Co_ord_X, message are encoded into AAL-DATA-req and decoded from AAL-DATA-ind.

3.2. Process Q_2931_X

The process Q_2931_X deals with connection establishment, maintenance and clearing. For every signalled ATM connection, a separate process will be instantiated. The process Q_2931_X has 11 control states[6]:

Figure 2 shows the state transitions of the Q_2931_U process. The sending of messages is denoted with "!" and the reception with a "?". The starting (Txxx) or the resetting (Rxxx) of timers are shown below the actions. The following simplifications are used to make this overview succinct:

- Before sending a message an appropriate message primitive from applica-

[6] The terms incoming and outgoing are always from the users point of view at the UNI.

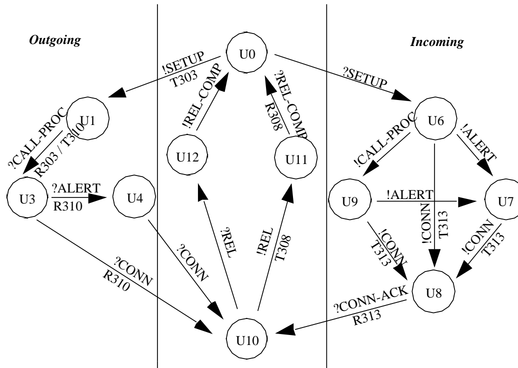

Fig. 2. Transitions of Process Q_2931_U.

tion_U has to be received. The reception of the primitive is not shown in the diagram.

- Usually, the reception of message is followed by the notification to the application_U with the help of a corresponding primitive. The sending of this primitives is not shown in the diagram.
- The time-out of a timer, except in *U1*, where a time-out is followed possibly by a retransmitted SETUP, is always followed by a call clearing, i.e. termination of the call controlling process after notifying the application. These transitions are not shown in the diagram.

3.3. Processes Reset_Start_X and Reset_Response_X

Like the process Co_Ord_X, the processes Reset_Start_X and Reset_Response_X are instantiated during system instantiation. The RESTART procedure can be executed at any time and is used to return virtual channels controlled by a peer entity back to idle state and to terminate the respective Q_2931_X processes. The following description outlines the behaviour of both processes.

The process application_X may enforce process Reset_Start_X in state *Rest-X0* to send a RESTART message, to start timer T316 and to change to state *Rest-X1*. RESTART is retransmitted after expiry of T316, at most *RSN* times. *RSN* is implementation-dependent. After receipt of RESTART-ACKNOWLEDGE in *Rest-X1*, the process resets timer T316, notifies the application and changes to state *Rest-X0*.

The process Co_Ord_X delivers a RESTART message to the process Reset_Response_X, which in turn changes to state *Rest-X2* after notifying the application_X and starting timer T317. After successful restart acknowledgment by the application, the process Reset_Response_X cancels timer T317, sends a RESTART-ACKNOWLEDGMENT back to the peer entity, and returns to *Rest-X1*.

The separation into two processes for restart handling has the advantage that RESTART requests issued simultaneously by the user- and the network side can be processed separately. Therefore, the user- and the network side have the possibility to release committed resources independently.

3.4. Specification Decisions

Several specification decisions have been taken to enable the validation of the above mentioned error classes, to reduce the system state space and to generate a validator of manageable size.

For validation class 5. ('loss of synchronization') errors, both sides are connected via a Signaling ATM Adaptation Layer - SAAL ([ITU95]/8.2 and [ITU94]). SAAL offers the following eight primitives:[7]

- AAL-ESTABLISH-request/-indication/-confirm
- AAL-RELEASE-request/-indication/-confirm
- AAL-DATA-request/-indication

Roughly speaking, SAAL distinguishes between two states: *idle* and *connected*. In addition, a spontaneous reset of the SAAL in the state *connected* was specified to indicate its reset with AAL-ESTABLISH-ind.

Since for every call an independent Q_2931_X process is started, only one call at a time is supported to limit the system state space.

Signals from the application or the SAAL, which have to be distributed to all Q_2931_X processes, use only commented output symbol in the SDL description of [ITU95]. This incomplete specification for signal distribution has been made explicit: The transitions were redefined, so that it is first checked that a Q_2931_X process exists and afterwards it is checked that the process is not already terminated. Only if both checks are positive, the signal is sent to the respective process.

A data type (Message) is used to declare all possible messages of the AAL-DATA primitives with their parameters. ASN.1 would be needed to declare a data type that consists of different alternative data structures with optional fields. However according to [ETSI95], ASN.1 shall not be used for validation purposes. Therefore, the Message data type integrates all necessary information elements needed for ATM signalling.[8] Furthermore, a verification to check the correctness of information element values is performed only for those information elements, which directly (e.g. CALL-STATE, CAUSE) or indirectly (e.g. CALLREFERENCE) influence a transition.[9]

[ITU95] uses the implementation-dependent procedure Verify_Message to examine incoming messages from the peer-entity for protocol conformance. The result may take the following values from type ResultType: RAP (Report and Proceed), OK (OK and Proceed), CLR (Release Call), RAI (Report and Ignore), and I (Ignore). For validation purposes, every possible result of Verify_Message should be executed at least once. To control the result of Verify_Message, the request-primitives of the environment are parameterized with a new parameter ACTIONIND of type ResultType. It is used to control how a receiving peer-entity should handle a received message. Besides the introduction of this new parameter, additional smaller changes in the specifications were necessary. The changes shall be illustrated with the help of the Fig. 3. The signal Notify_req_1

[7] The primitives AAL-UNIT-DATA-request/-indication were not specified, because [ITU95] does not use them.

[8] In the following, the contents of an information element is referenced by the variable v.

[9] For example, the information element "ATM Traffic Descriptor" (ATD) was not checked, because it is delivered to the user.

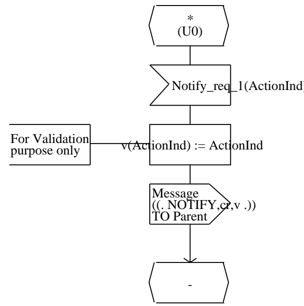

Fig. 3. Modification for validation.

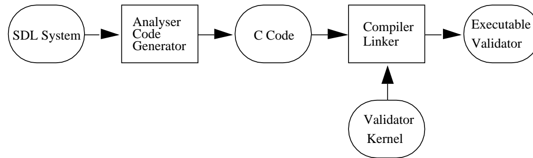

Fig. 4. Generation of a SDT Validator [SDT31].

has a new parameter (ActionInd). The parameter is used to change the variable list v belonging to every message. The element ActionInd of v is set to exactly this parameter. In the procedure Verify_Message, the receiver reads this variable and makes its decision with respect to the variable content. The default value of ActionInd is OK. Another possible solution for specifying non-determinism related to the Verify_Message result would have been the introduction of a non-deterministic decision with the any construct instead of a controlled and deterministic one. However during state space exploration, every possible result of Verify_Message would be analyzed, which results in a state space explosion. Therefore, the first approach was taken so that it allowed us to model the grade of non-determinism explicitely in order to control the size of the system state space.[10] Subsequently, the term non-determinism refers to this approach.

4. Validating the ATM Signalling Specification

Based on the general goals for validation discussed in Section 2, the following concrete goals for validation have been addressed in the case study. At first, experiments for the model-specific error classes were made (classes 1 to 4). At next, protocol specific error classes were examined (classes 5 to 7). The experiments for the model-specific error classes were performed using automated reachability-analysis algorithms described above. The experiments for protocol specific error classes were performed (semi) automatically.

The SDT validator [SDT31] has been used for automated reachability analysis. The process of generating the validator is shown in Fig. 4. A C program is derived

[10] Another reason for selecting the solution with explicitely controlled deterministic decisions was that the simulation kernel of SDT 3.1, which is the core of the used validator, can not handle non-deterministic decisions.

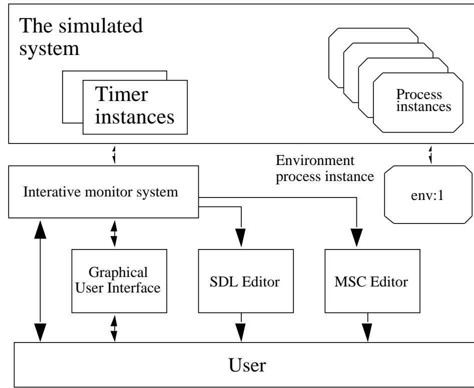

Fig. 5. Structure of a validator/simulator [SDT31].

from the SDL specification and is made executable by linking it to the specific run time library for validation purposes, the so called validator kernel.[11]

4.1. Structure of the Validator

The generated validator consists of the following components:

- the system, which has to be validated (the executable application),
- the environment process,
- the interactive monitor system, and
- the graphical user interface (the so called validator user interface).

In addition, the SDL editor and the MSC editor are used at run-time and serve to graphically trace and animate executed transitions and sent and received signals, respectively. A validator contains several data objects, which represent the different SDL objects like process instances, signal instances, etc. Together with the process instance env:1, representing the system environment, and the monitor system they make up the validator (Fig. 5). The validator can be used stand-alone or in combination with the graphical user-interface.

4.2. Validation Principles

The SDT validator is based on the reachability analysis discussed in Section 2. It uses directed graphs to represent system execution, which are called *behaviour trees*. The nodes of such a behaviour tree represent SDL system states. A SDL system state consists of the control states per process, their waiting queues, their variables, the status of procedure calls (incl. local variables), and active timers.

Branches of a behaviour tree represent single SDL events, which bring an SDL system from one system state to another. Depending on the validator's configuration the SDL events can be single SDL statements like **task**, **input**,

[11] SDT supports several run-time libraries for different purposes. For example, a simulator can be generated with the simulator kernel.

output, etc. or complete SDL transitions. According to this setting, the size of the generated behaviour tree varies.

The set of all system states, which can be represented by a behaviour tree is called the *system state space*. While traversing the tree, the behaviour of the SDL system can be examined. In addition, the system states can be examined whether specific criteria such as the presence of a deadlock are fulfilled. The state exploration can be executed either manually or automatically.

4.3. Random-Walk Analyses, Bit-State Exploration and Reports

SDT offers two different automated validation methods, i.e.

- a depth-first bit-state exploration and
- a random-walk analysis.

In a bit-state exploration, the validator follows the behaviour tree depth-first until either a rule is fulfilled, the maximum search-depth is reached or an already analyzed state is reached. All possible paths of the tree are walked through. On the other hand, the paths through the behaviour tree are chosen randomly in a random-walk analysis.

During a manual or automated state space exploration, a set of rules is validated in every state to identify predefined system conditions such as errors or selected system states. Whenever a rule is fulfilled, it triggers the generation of so called user reports.

A report contains the information about the system state, in which the rules is fulfilled. The validator can be configured such that in an automated state space exploration

- the fulfillment of a report will be reported to the user and the search will be aborted (*Abort*),
- a report will be generated and the behaviour tree below this state will not be further examined, i.e. the search will be pruned (*Prune*), or
- only a report will be generated without affecting the search (*Continue*).

In the case study, all three possibilities are used. User reports can be used to track a problem. Predefined rules exist for Deadlock detection, Loop detection, ImplSigCons(umption) detection, ChannelOutput (Error) detection, MSC Verification and others.

The rules ‘Deadlock’ and ‘Loop’ are validator internal mechanisms to check, whether a deadlock or a livelock occurred. They can be use to validate errors of classes 1 and 2. Both rules were configured, so that the state space exploration is aborted on their fulfillment. Deadlocks and livelocks are considered to be serious errors and hence, such a misspecified protocol is not further examined.

With the help of the rules ImplSigCons and Channel Output error class 3, i.e. Reception Errors, can be validated. An ImplSigCons occurs if a signal is sent to a process, which cannot handle or save the signal in its current state. It is therefore implicitely consumed. In this case the protocol is underspecified. Channel Output errors occur if for instance a signal is sent to more than one receiver or to an already terminated process. Such an error situation hints at a potential protocol design error.

Table 2. Random-Walk (RW) – state space generation options.

Parameter	1st Experiment	2nd Experiment
Scheduling	All	All
Transition	SDL	SDL
Priorities	1 2 2 2 2	1 2 2 2 2
Repetitions	20000	20000
Search-depth	200	200
TestValue	ResultType={OK}	ResultType= {OK,RAI,RAP,I,CLR}

5. Validation Experiments

This section describes the executed experiments and their results. All experiments were executed on one machine to obtain comparable results with respect to the measured validation time. The experiments were carried out on a SUN ULTRA I CREATOR 3D (200MHz, 256 MB).

5.1. Random-Walk

The first experiment was a random-walk (RW) experiment in order to get an overview of the size of the generated state space. To get detailed insights into the protocol, two experiments were performed.

The first one with a very small state space and the second with a state space as big as possible. 'Small' and 'big' refer to the possibility of modeling non-determinism. If including non-determinism, the state space will get much bigger than without. The following parameters were set:

The SCHEDULING parameter in Table 2 defines, which process instances of the ready-queue are selected for further execution. The ready queue is a queue, which contains all process instances that received a signal and are ready to execute a transition. The ready-queue is sorted by priority and insertion time. The value *All* defines that all process instances in the ready queue can be executed. This complies to the semantics of SDL and is therefore chosen.[12]

The TRANSITIONS parameter in Table 2 allows one to give the transitions in the behaviour tree different interpretations: The value *SDL* defines behaviour tree transitions to be complete SDL transitions, e.g. from a beginning process control state to an ending process control state.[13]

The PRIORITIES values in Table 2 describe the priorities for internal events, inputs from environment, time-out events, channel outputs and spontaneous transitions, respectively. Valid values for these parameters range from 1 to 5 with 1 having the highest priority and 5 the lowest. The chosen values reflect the idea that before other actions are possible all internal actions should have been finished first. This allows maximal internal progress of processes, e.g. without any interaction with other processes. All other actions have the same priority.

[12] The alternative value *First* would define that only the first process instance being in the ready-queue is selected for execution. This complies for example with the semantics of a SDL prototype that can be generated with the SDT C-Code generator.

[13] The value *Symbol-Sequence* defines a behaviour tree transition to be the longest possible sequence of SDL symbols that can be executed without any interaction with other processes.

Table 3. Results of the random-walk experiments.

1st Experiment	2nd Experiment
No of reports: 82	No of reports: 91
Gen states : 14207031	Gen states : 33762148
Max depth : 200	Max depth : 200
Min depth : 16	Min depth : 15
Symbol coverage : 77.05	Symbol coverage : 82.90
Validation time: 5682 s	Validation time: 13504 s

The SEARCH-DEPTH parameter in Table 2 defines the depth at which the state space exploration should be aborted. It was set to 200. Pre-trials showed that neither an increasing coverage or the creation of new reports can be realized with an increased search-depth above this limit.

The random-walk analysis can be parameterized with the REPETITIONS parameter in Table 2. This parameter states how often a search through the behaviour tree will be performed. The more often a path through the behaviour tree is searched the higher is the possibility to detect an error. Pre-trials showed that a value of 20,000 seems to be high enough since the symbol coverage didn't increase with an increased repetitions parameter. The symbol coverage converged to 83%.[14]

In the first experiment, only those signals from the environment were permitted that trigger an OK in the Verify_Message procedure of the receiving peer entity. Hence, in this experiment no behaviour of the ATM signalling protocol was validated, which required the sending of STATUS messages. Only the valid behaviour of the protocol was examined. The 2nd experiment included signals from the environment, which could trigger every possible result of the Verify_Message procedure at the receiving peer entity. It was expected that the symbol coverage of the 1st experiment would be smaller than the symbol coverage of the 2nd experiment.

The SDL specification of [ITU95] deliberately uses *implicit signal consumption* in the SDL description of the protocol. Only those transitions are defined that have an effect on the protocol behaviour, i.e. transitions for simple signal consumption are not specified explicitely. Therefore, it is assumed that an implicitely consumed signal is not an error but rather part of the protocol. For this reason, the action for report type ImplSigCons is defined to be Continue. This has the disadvantage that every report of type ImplSigCons has to be evaluated manually, because it could not be excluded that 'real' protocol errors were found and reported.

Table 3 gives an overview of results of the first two experiments. The search depth of the state space exploration can clearly reach depths above 100. This result is the basis for some thoughts in dimensioning the below described bit-state-analysis experiment, where the search depth was limited to 100. The generated reports of the experiments are discussed in Section 6.

[14] The coverage is discussed in detail in Section 6.2.

Table 4. Bit State (BS) - state space generation options.

Parameter	Bit State Exploration
Scheduling	All
Transition	SDL
Priorities	1 2 2 2 2
Hash-size	200 MByte
Search-depth	100
TestValue	ResultType= OK

5.2. Bit-State Exploration

Because of the system complexity, system state space cannot be analyzed completely with a bit-state exploration. Therefore, the system state has to be narrowed. For the bit-state exploration experiments, the relevant parameters were chosen as given in Table 4.

The parameters Scheduling, Transition, Reports and Priorities were identical to the random-walk experiments. The main point is the limitation of the state space in order to get validation results within a feasible amount of time (e.g. within days and not weeks or months).

In a bit-state exploration, the already analyzed system states are stored in a hashing-table. The size of this hashing-table is very important, because whenever a system state is analyzed by comparing its hash-value with already existing hash values a collision risk exists. The collision risk is measured in percent and should not exceed 1% [SDT31]. The collision risk is inversely proportional to the size of the hash-table, i.e. it decreases if the size of the hash-table is increased. The collision risk is reported by the validator after finishing a bit-state exploration.

In the case study experiments a hash-table size of 200MB was chosen. A larger size would force the computer on which the validation is running to use the swap-space of memory on the hard disk.[15]

The pre-trial of the bit-state experiment showed that bit-state exploration can last many hours. Therefore, the search-depth was limited to 100. After completed exploration, the collision risk was stated to be 1%, i.e. being in the acceptable limits. The TestValues were limited to OK. Pre-trials showed that if admitting all possible values of type ResultType, the collision risk increases up to 15% at a search depth of 100. Table 5 gives an overview of the bit-state analysis. A detailed discussion of the results is presented in the next section.

6. Validation Results

Before a detailed analysis of the reports is given, the results of random-walk and bit-state analysis are compared. Comparing the first random-walk experiment, i.e. small state space, with the bit-state analysis, the following should be remarked:

- the amount of generated reports is the same,

[15] This would lead to an increase of the validation time by a factor of at least 100 (under the assumption that the mean access time for the main memory is 100 ns and that the mean access time for the hard disk is 10ms). Therefore, bit-state explorations with larger hash-table sizes were not executed.

Table 5. Results of the bit-state-exploration experiment.

** Bit state exploration statistics **
No of reports: 82. Generated states: 512163204. Truncated paths: 90574771. Unique system states: 347226652. Size of hash table: 1600000000 (200000000 bytes) No of bits set in hash table: 559978441 Collision risk: 1% Max depth: 100 Current depth: -1 Min state size: 860 Max state size: 1376 Symbol coverage: 77.05 Validation time: 204865 s

Table 6. Amount of reports per error class.

No.	1st RW	2nd RW	BS	Total
Class 1.	20	20	19	59
Class 2.	24	26	26	76
Class 3.	16	8	15	39
Class 4.	6	10	6	22
Class 5.a	2	0	2	4
Class 5.b	2	4	2	8
Class 5.c	0	9	0	9
Class 5.d	0	3	0	3

- the coverage is the same, and
- the validation time needed to run the bit-state analysis is 36 times greater than the time needed for the random-walk.

As described below, different types of errors can be isolated from the reports. Table 6 gives a comparison of the quality of both methods with respect to the detection of protocol errors and problems under certain conditions.[16]

First of all, the comparison between the bit-state exploration and the first random-walk is reasonable, because it can be assumed that the bit-state analysis has found every error within a search depth of up to 100. For every error class reported using the bit-state approach, the random-walk approach also produced a report. Therefore it can be shown that the random-walk has comparable error coverage, in particular since the detection of one error in an error class is often enough to fix the problem.

On the other hand, the first random-walk experiment searched to a depth up to 200, but could not find any new error classes.[17] Therefore it can be assumed that the search depth of the bit-state analysis was well chosen.

The second random-walk using an extended state space detected errors of two new error classes. However, no class 5a type reports were generated.

---

[16] The error classes are explained in detail in Section 6.1.2.
[17] The additional reports were not on any new protocol error.

Table 7. Generated reports.

Experiment	Report type	Number
1st Random-Walk	Create	6
	Output	6
	ImplSigCon	70
2nd Random-Walk	Create	6
	Output	5
	ImplSigCon	80
Bit-State	Create	6
	Output	6
	ImplSigCon	70
Total		255

It should be noted again that due to the nature of validation, it cannot be stated that all errors were detected. It is possible that other, undetected errors exist in the protocol.

6.1. Reports

This section discusses the generated reports in more detail. While running the three validation experiments 255 reports were generated (see Table 7). Deadlocks or livelocks were not found in the protocol.

6.1.1. Create and Output Reports

The reason for the generation of a create-report is the attempt to instantiate another instance of a process although the maximum number of instances of this process already exist. As explained in the specification decisions in Section 4, in this model, only one instance of every Q_2931_X process is permitted. If the Co_Ord_X process receives a Setup_req from its application, while another call is already in progress it tries to instantiate another instance of the Q_2931_X process, which results in the generation of a create report.

Related to this problem is that after an attempt to create a Q_2931_X process, a Setup_req_1 is sent to this process. Because the creation was not successful, there is no receiver for this signal. Hence, an output report is created. These reports are generated due to limitations of the SDL specification used. They do not represent protocol errors.

In the following, we discuss the causes for generation of the create and output reports:

1. For any implementation of this protocol, it is essential to define an upper bound for the number of calls being controlled at one time. The protocol does not provide the capability to signal the user that no further calls can be established. The user has to detect the lack of a Setup_conf, Release_conf, etc. He is obliged to implement a self-defined timer to be able to detect the absence of Setup_conf, etc.
2. A call that is in the call-establishment phase or in the active state of Q_2931_X will not be notified of a link release initiated by an application (with Link_Re-

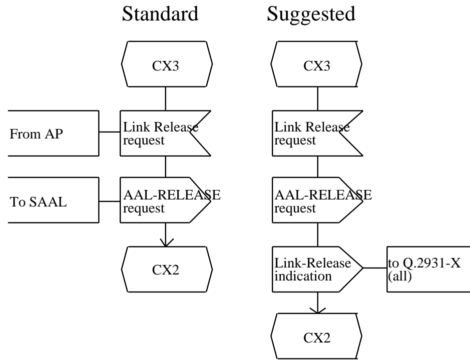

Fig. 6. Suggested enhancement 1.

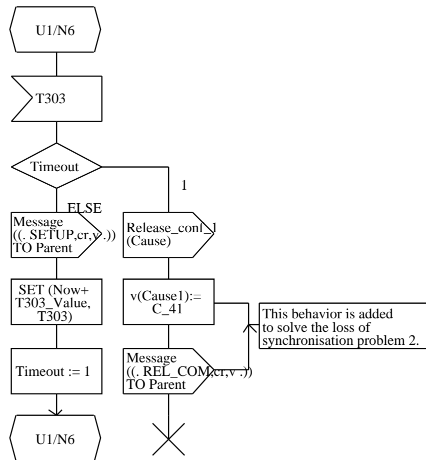

Fig. 7. Suggested enhancement 2.

lease_req) on its side of the UNI. Therefore, it continues to exist although the underlying SAAL to the peer-entity has been released. On the other hand, the process Q_2931_Y at the peer entity will be handled correctly as defined in [ITU95]/5.6.10, i.e it will be terminated. Hence, loss of synchronization occurs, i.e. on one side an instance for a call exists while on the other not. Therefore, an enhancement to the protocol as shown in Fig. 6 is suggested.

3. An entity that receives a SETUP message from the peer-entity for a call-establishment, first instantiates Q_2931_X process. Then, this process notifies the application with a Setup_ind. If for any reason the application does not respond to the Setup_ind, since for example the application is unable to respond due to overload, the SETUP message will be retransmitted from the initiating entity. The process Co_Ord_X will classify the SETUP as a retransmission and will ignore it. Therefore, the Co_Ord_X will not notify the application that an unanswered call exists.

A first question is, why should a retransmission take place since a SAAL connection is assumed to be reliable, i.e. an assured and reliable transport channel exists. The need for retransmission is caused either by the protocol

entity or the application and not by the SAAL connection, so that in both cases a retransmission will not show the desired result. The retransmission is not the main issue here. On an unsuccessful retransmission, i.e the peer entity does not receive any response to its SETUP, the call will be cleared locally at the peer-entity, i.e. only the application will be notified with a Release_comp about the unsuccessful call establishment. Again, loss of synchronization is the result. One peer entity will terminate the associated instance of process Q_2931_X, the other one will not and still occupies resources. In this case, the process instance for the call can only be terminated with the restart procedure, since none of the peer-entities are aware of this process instance. A possible solution is that as in all other states, the peer-entity informs the other peer-entity about the call-release. A message, RELEASE_COMPLETE with cause 41 "temporary failure" is suggested. The suggested enhancement is presented in Fig. 7.

6.1.2. Implicit Signal Consumption

The majority of reports were generated on implicit signal consumptions. The following error classes could be extracted from these reports:

1. Application_X sent unexpected primitives to Co_Ord_X, Q_2931_X as well as to Reset_Start_X and Reset_Response_X in a certain state.
2. In a certain control state, signals are not being expected that come from another process of the same entity as the direct reaction to the reception of a signal.
3. Consecutive errors, which occur due to the non-notification of a Q_2931_X process of a link-release initiated by its own application_X. See also comment 2. in Section 6.1.1.
4. The protocol does not deal with the question in which state the process Reset_Start_X respectiveley Reset_Response_X have to be in the case of a link-release while a restart-procedure is in progress. Implicit signal consumptions in state CX0 of process Co_Ord_X is the result.
5. Messages of the peer-entity are not expected. In detail
   (a) Retransmission of RELEASE is not expected at the peer-entity.
   (b) Messages of the peer-entity can be received in states other than the desired one.
   (c) The effects of ignoring a message is not clear (Result = I after Verify_Message).

The reports can be explained as follows.

to 1): The specification does not specify the reception of signals that do not influence the protocol behaviour and can be ignored, i.e. are of no interest. They form an underspecification in a very narrow sense. According to the SDL semantics, the reception of those signals is implicitely defined. However, this is not considered to be a good specification style. The validator addresses this fact by creating warnings and not errors. Obviously, this seems to be a specification decision taken during design of [ITU95]. Therefore, no attention was paid to these reports.

to 2): If a process instance A (e.g. Co_Ord_X) in state *1* (e.g. *CX3*) sends a signal (e.g. Link_Release_ind) to an other process instance B (e.g. Q_2931_X) of the same entity and the answer from B in the new state *2* (e.g. *CX0*) of process instance A is not expected, then this signal will be implicitely consumed. In contrast to 1) were the signal is not dictated by the protocol, the reception in this case should be specified, because the signal has to be sent and therefore a transition should be defined for it. Hence, the reception of these signals were specified in appropriate transitions.

to 3): This problem was already discussed in the context of create-reports in the previous section, and therefore the suggested solution was applied.

to 4): [ITU95] does not deal with the problem, what happens if the loss of the SAAL-link to the peer is indicated, while a restart-procedure is running, i.e. while the processes Reset_Start_X or Reset_Response_X are in state *Restart-Request* or *Restart-Response*, respectively. For the entity, which initiated the restart-procedure, this means that a needed retransmission of a RESTART message would not be possible, because process Co_Ord_X is in state *CX0*. Also, a RESTART-ACKNOWLEDGE message cannot be received. The specification defines that after an implementation-dependent number of retransmissions the restart-procedure is aborted. A Reset_error_ind_1 for notification will be send to the application. If the application tries to initiate another restart-procedure, the SAAL will be re-established first. The peer-entity should respond to a RESTART message with a RESTART-ACKNOWLEDGE message. It would be sufficient to specify the reception of the RESTART-ACKNOWLEDGE in state *CX0* of process Co_Ord_X. If the process Co_Ord_X receives the primitive Reset_error_ind in state *CX0*, it should be forwarded to the application.

A possible solution is to specify for CX0 appropriate transitions for the reception of RESTART and RESTART-ACKNOWLEDGE messages and for Reset_error_ind primitives coming from the process Resert_Start_X. Except for forwarding a Reset_error_ind to the application, no further actions are needed. Similar arguments as discussed in 2) apply in this case. Consequently, the consumption of these signals is specified explicitely in the enhanced specification.

to 5a): If a RELEASE message was received correctly at the peer entity, which is waiting for the corresponding answer from the application, the reception of an additional retransmitted RELEASE message according to [ITU95]/5.4.3 is not specified. However according to [ITU95], a reception of a RELEASE message in state release-request (*N11*/*U12*) has to be specified. On the other hand, [ITU95] does not mention, whether an application has to be notified about the reception of a second RELEASE message. It is concluded from the fact that a second optional cause information element can be added to a RELEASE message (i.e. also to an additional second RELEASE message). The SDL-specification was changed accordingly.

to 5b): Two situations exist in which a process (Reset_Start_X and Q_2931_X) can change its state while waiting for an acknowledging message from the peer entity (time-out and call-release, respectively). In the respective subsequent states, the reception of those messages, which are possibly already in transit, is not specified. The SDL specification was extended accordingly.

to 5c): [ITU95] defines that after receiving a message its contents should be verified. As a result of this verification, a message, for instance a faulty one, may be ignored. When messages are ignored, the following problems exist:

- How should a process Q_2931_X being in state *X0* be terminated after a SETUP message has been ignored? In this state, neither a timer is running nor the application can be informed nor a retransmitted SETUP will reach the process (see explanation above). A Release-procedure cannot be invoked, because in this state the reception of RELEASE messages or Release-req primitives is explicitely excluded. Further, a status-enquiry procedure is not possible. Hence, the call-establishment can neither be continued nor be aborted with RELEASE or RELEASE-COMPLETE messages.
- On the other hand, for all states of Q_2931_X processes different than the *Null* state, after ignoring a message the status-enquiry procedure could be invoked to identify in which status the call is. However, the textual description of the protocol gives no clue what really happens if a message is to be ignored. Such situations in the protocol are not described unambiguously.

A complete solution to this problem could not be found, because the motivation for the possibility of ignoring messages is unclear. However, there is a need to forbid 'silent' message consumption of invalid messages at least in state *Null*. The specification was changed in such a way that if the procedure Verify_Message returns I (Ignore) in state *Null*, the same behaviour applies as if the procedure had returned CLR (Clear Call).

to 5d): For state *N11/U12* is specified that if the procedure Verify_Message returns CLR (Clear Call) as result, the process should change to *N0/U0*. This is wrong according to [ITU95]/5.6.12. It is stated that if the CallState of a STATUS message equals *Null*, all resources should be deallocated and the process should return to the *Null* state. Here, an ambiguous meaning of state *Null* exists. State *Null* in the textual description with respect to SDL means that no process exists for a call. That implies also that if all resources should be deallocated this process has to be terminated. On the other hand, process Q_2931_X has a state *X0* (*Null*). This state must not be mixed with state Null from the textual description. However, that happened in the SDL specification and resulted in the report. Therefore, the specification was changed so that, if the procedure Verify_Message returns CLR (Clear Call) as result the process will be terminated.

6.2. Unreachability in the Protocol

The following section examines, whether the protocol contains unreachable parts, i.e. error class 4 of the model error classes. The discussion is based on the results of the second random-walk, because this is the one which deals with the largest state space. Only the symbol coverage will be discussed, as it allows for the analysis of unused alternatives in decisions during the course of the state space exploration. At the end of the experiment, the symbol coverage was 82.9%. The whole system consists of 1152 symbols and 197 symbols were not reached.

The first step in analyzing this random-walk was to examine, whether all unreached symbols are indeed unreachable. This was done by manual state space exploration and was based on protocol knowledge. Through manual revision of the SDL specification, the coverage could be increased by 10% (111 symbols)

to 92.97%. Only 86 symbols were left. It should be mentioned that to achieve this increase, it was necessary to change some parameters of the state space exploration algorithm (priority, symbol time). Due to the need for limiting the state space internal events got the highest priority. Doing so prevented time-outs being executed, if the behaviour of a specific feature did not need any communication with the environment (e.g. STATUS-ENQUIRY-procedure). The coverage after manual revision forms the basis for the following discussion.

The remaining 86 symbols were classified according to the following. The examined system contains parts that are not specified in the protocol description and their examination is not within the scope of this work (e.g. SAAL, Check_Msg, etc.) Unreached symbols in this category should not be included in coverage discussions. There were 26 from the 86 (30%) symbols. Therefore, only 60 symbols from the total number of protocol specifical symbols (1552) were really unreachable (5.25% of 1152).

Due to shortcomings of the validator, which cannot handle non-determinism, no possibility was found to tag STATUS messages with rearranged states or causes. For this reason, not all alternatives of decisions were enabled for execution. Only the alternatives of the default values were validated. For the same reason, no possibility was found to send messages, which are not included in the protocol description.

These two groups consist of 54 (34 STATUS- and 20 other messages) of the remaining 60 symbols, i.e. 90%. The last 6 symbols could not be reached due to implementation decisions. For example, an implementation either supports a recovery option or not. Hence, some specified symbols are not executable.

7. MSC Validation

Message Sequence Chart (MSC [ITU96]) validation, i.e. validating requirements that are specified as MSCs, was used to investigate properties of the protocol, which have not been addressed by the experiments described above. Two requirements, which could have been formulated during the developing process for the protocol, were taken and validated against the SDL specification.

1. A user initiates a call-establishment with a Setup_Req. First, he gets an acknowledgment from the network (Proceeding_ind) followed by a Setup_conf indicating the completion of the call establishment.
2. A protocol entity of a call being in state Active and receiving a AAL_ESTABLISH_ind from the SAAL has to invoke a status enquiry procedure by sending a STATUS_ENQ to the peer-entity. The peer-entity has to respond with a STATUS message containing its state and as cause “Response to StatusEnquiry” (cause=30). If the peer-entity is in a compatible state, than the call has to remain in state Active. A status-enquiry procedure invoked by the peer entity at the same time has to be handled in the same way, i.e. by returning a STATUS message containing the call state of the entity and the cause “Response to StatusEnquiry”.

These two informal descriptions were transformed into formal ones. The respective MSCs are shown in Figs 8 and 9. The ATM signalling protocol should be examined whether it shows the desired behaviour as described in these MSCs.

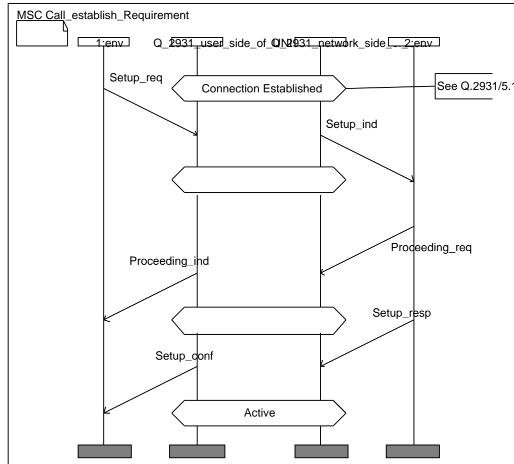

Fig. 8. 1st protocol requirement as an MSC.

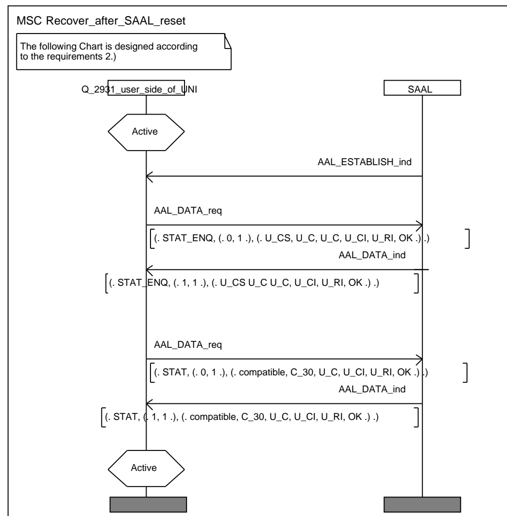

Fig. 9. 2nd protocol requirement as an MSC.

While transforming the informal descriptions into MSCs, some peculiarities have to be paid attention to. For instance, the first requirement names only one user of the service (here 1:env). This user interacts with the protocol entity Q_2931_user_side_of_UNI. A one-to-one mapping into an MSC would have to use only the two left process instances in Fig. 8 (1:env, Q_2931_user_side_of_UNI) and their signals. SDT requires in an MSC, which is used for validation, always the complete signal exchange between all involved processes. Therefore, the description of the network side has to be part of the MSC.

The MSC in Fig. 9 is not an exact representation of the 2nd requirement. Only one possible trace is shown. This is due to the fact that SDT supports only a subset of MSC92 let alone MSC96 and operators like alt cannot be

Table 8. MSC validation results.

	1st Experiment	2nd Experiment	3rd Experiment
No of reports	0	2	5
Generated states	30	34	781171
Truncated paths	0	0	214
Unique system states	29	31	160130
Size of hash table	10000000 bytes	10000000 bytes	10000000 bytes
No. of bits set	58	63	319535
Collision risk	0%	0%	0%
Max depth	28	28	1000
Current depth	28	28	23
Min state size	924	1140	1140
Max state size	1280	1356	1452
Symbol coverage	16.40%	21.52%	23.28 %
Reports		1 MSC Verification 1 MSC Violation 3 Impl. Sig. Cons.	1 MSC Verification 2 MSC Violations

used. Therefore, several MSCs are needed to describe the 2nd requirement. Three experiments on an MSC validation were executed, one on the first requirement (small state space) and two on the second requirement (small and big state space). The first two experiments are used to validate that the protocol is able to behave as defined in the 1st and 2nd requirement, respectivly. The third experiment was run to observe the influence of the state space size on the MSC validation. To obtain a small state space the Priorities parameter was set to "1 2 2 2 2". The state space was enlarged in the third experiment, with the setting Priorities parameter to "1 1 1 1 1". To enlarge the state space, all SDL events were handled with same priority in the third experiment (Priorities="1 1 1 1 1"). Please refer to Section 5.1 for a discussion of these parameters.

After the protocol was set to an appropriate starting state, either manually or with user-rules, the MSC validation was started. Both MSCs were validated. i.e. the protocols meets the requirments. As a result of an MSC validation, reports of type "MSC verification" were generated. These reports include detailed MSC, which describe the complete signal exchange between all participants. The generated reports can be summarized as described in Table 8. Please note that the MSC violations stated in Table 8 only reflect possible alternative executions of the protocol, which are in contradiction to the required behaviour.

Essential for the MSC validation is fulfillment of the MSC by at least one execution sequence of the protocol, what is comparable to the eventual operator in temporal logics.

8. User Rule Validation

Finally, validation experiments on the basis of user rules were executed. The aim of these experiments were to validate

- if all valid state combinations between user-and network side can be reached and
- if forbidden state combinations ("loss of synchronization") occur.

Table 9. User-rule state space generation options.

Parameter	Bit State Exploration
Scheduling	All
Transition	SDL
Priorities	1 2 2 2 2
repetitions	10000
search-depth	100
TestValue	ResultType= {OK}
Report	User-Rule=Abort

34 valid system state combinations were identified. Eleven system states for incoming and outgoing calls and twelve for the call release. As in the specification, we distinguish between outgoing, incoming, and calls being released.

MSCs cannot be used for describing these state combinations, because "local conditions" on block level are not supported by the validator. User-rules can be used to formulate such state combinations. The user rules are validated with a random-walk state space exploration, i.e. whether these state combinations are reachable or not[18]. Before starting the user rule validation, the validator was placed in a system state with an established SAAL. This reduces the amount of time needed for the validation. The parameters for these experiments are given in Table 9.

The user rule validation is exemplarily illustrated for the state combination *U3* and *N4*:

- With the help of the SDT navigator, which allows one to bring interactively and step-by-step the validator into a certain system state, the validator was placed in the SAAL Connection Established state.
- User-rule was defined:

```
define-rule (state(Q_2931_U:1) = U3 and
             state(Q_2931_N:1) = N4)
```

- A random-walk analysis was run.

All defined system state combinations could be validated. The validation time varied from less than one second to up to half an hour.

A drawback of this method is that only one user-rule can be validated at a time. This lead to 34 separate validation experiments. The concatenation of the individual user rules with the help of logical OR operators was not useful, because every generated report has to be analyzed to identify, which state combination was reached. The following state combination defines a loss of synchronization: "Loss of synchronization occurs if one side is in state active while the peer-entity is in state call present." This is described by the following user-rule

```
define-rule ((state(Q_2931_N:1)=N10)
        and not (exists U:Q_2931_U))
        or
        ((state(Q_2931_U:1)=U10)
        and not (exists N:Q_2931_N))
```

[18] A random-walk is sufficient if and only if the occurrence of a certain state combination can be found.

Table 10. Results of the final random walk.

Experiment with the Enhanced Specification	
No of reports:	59
Gen states :	34522281
Max depth :	200
Min depth :	15
Symbol coverage :	85.32%
Validation Time:	13808 s

The user-rule was checked with the bit-state exploration described above. No loss of synchronization was detected, but that does not mean that this combination cannot occur. It can only be stated that, under the above described conditions, up to a depth of 100 this state combination does not occur.

It should be noted that user-rules in the current release of SDT cannot use their whole power of expressiveness. For example, a complete description for loss of synchronization is the negation of all valid state combinations identified above.

However, the corresponding user-rule

```
define-rule NOT(SC1 OR SC2 ... OR SC34)
```

is too long for the validator to parse. It consists of roughly 1700 characters, but the maximum for the length of a user rule are 1000 characters. Attempts to compress the size lead to an input line of roughly 1200 character, which was still 20% too long.

User-rules cannot only be used to check certain conditions, they can also be used to narrow the state space in dependency of system variables. A problem is that user-rules are not able to handle self-defined data-types, e.g. the following user-rule cannot be used

```
siexist(S:Setup_req-Co_Ord_U | S->1 = I)
```

9. Conclusions

With the automated validation of the ATM signalling protocol in [ITU95] and the analysis of the generated reports a number of errors and inconsistencies in the specification with respect to the textual description of the protocol were discovered.

A final random-walk with big state space has been executed on the corrected and enhanced SDL specification. We expected a smaller number of reports and neither additional errors nor new types of errors. This experiment was parameterized as the 2nd RW experiment described above. The results of the final experiment are given in Table 10.

From the 59 reports generated eight were create and output reports caused by the “one connection” restriction and therefore be ignored. The remaining 51 errors were divided into 43 errors from class 1 and eigth error from class 5c. With the used depth and repetition parameters, the final random walk shows a sufficient coverage of 83% with only 1% below the theoretically computed coverage of 84% (= 967 of 1152 symbols). Based on a manual reachability analysis of the remaining 17%, the protocol does not contain unreachable parts.

Besides the validation of the ATM signalling protocol, the paper describes

an exercise in automated protocol validation in general. Means for state space exploration, report generation, MSC validation and user rule validation allow one to validate a number of protocol properties. Of particular use are the MSC and the user rule validation, which can be used to describe in a simplified manner functional protocol requirements explicitely. However, complex protocol properties could not be validated due to certain limitations of the validator. For example, in an MSC validation, MSC concepts like coregions, subMSCs, inline expressions, or local and global conditions are not supported. The described work will be a base to further improve the tool support for protocol validation.

References

[ATM94] ATMForum: *ATM User-Network Interface Specification Version 3.1,* September 1994.

[Ek93] Ek, A.: *Verifying Message Sequence Charts with the SDT Validator*. In Ove Faergemand and Amardeo Sarma (eds.): *SDL '93 Using Objects Proceedings of the Sixth SDL Forum,* Darmstadt, Germany, 11-15 October, 1993, pp 237–249, North Holland.

[ETSI95] ETSI TC-MTS: Final Draft prETS 300414. Methods for Testing and Specification (MTS); *Use of SDL in European Telecommunications Standart. Rules for testability and fascilitating validation.* Sophia Antipolis, August 1995.

[Hol91] Holzmann, G. J.: *Design and validation of computer protocols* Prentice Hall, 1991.

[ITU93] ITU-T Recommendation Z.100: *CCITT Specification and Description Language (SDL),* Helsinki March 1993.

[ITU94] ITU-T Recommendation Q.2100: *Broadband ISDN - B-ISDN Signalling ATM Adaptation Layer (SAAL) Overview Description,* Geneva 1994.

[ITU95] ITU-T Recommendation Q.2931: *Broadband Integrated Services Digital Network (B-ISDN) - Digital Subscriber Signalling System No. 2 (DSS 2) - User-Network Interface (UNI) Layer 3 Specification for Basic Call/Connection Control,* Geneva Februrary 1995.

[ITU96] ITU-T Recommendation Z.120: *Message Sequence Chart (MSC),* Geneva 1996.

[Rud92] Rudin, H.: *Protocol development success stories: Part I.* In R.J. Linn, Jr., M. Uyar (eds.): *Protocol Specification, Testing and Verification, XII:* Proceedings of the IFIP TC6/WG 6.1 Twelfth International Symposium on Protocol Specification, Testing and Verification; 22-25 June 1992, Lake Buena Viesta, Florida, U.S.A., pp 149–160, North-Holland, 1992.

[SDT31] Telelogic Malm AB: *SDT 3.1 User's Guide, SDT 3.1 Reference Manual,* Telelogic, 1996.

[ScR96] Schieferdecker, I. and Rennoch, A.: *Validation and Correction of ATM Signalling Abstract Test Suite* ATM Forum Contribution, Orlando, June 1996.

[Wes92] West, C. H.: *Protocol Validation - principles and applications.* In Computer Networks and ISDN System, Vol. 24, pp 219–242, North-Holland, 1992.

[Zaf78] Zafiropulo, P.: *Protocol validation by Duologue Matrix analysis,* IEEE Trans Comm Vol. 26, pp 1187–1194, 1978.

Received March 1998

Accepted in revised form October 1998