# Dealing with the complexity of state space exploration algorithms for SDL systems

Jens Grabowski[a], Rudolf Scheurer[b],
Daniel Toggweiler[b], and Dieter Hogrefe[a]

[a]Medizinische Universität zu Lübeck, Institut für Telematik, Ratzeburger Allee 160, D-23538 Lübeck, Bundesrepublik Deutschland, e-mail {jens,hogrefe}@itm.mu-luebeck.de

[b]Universität Bern, Institut für Informatik, Neubrückstrasse 10, CH-3012 Bern, Schweiz, e-mail {scheurer,toggweil}@iam.unibe.ch

**Abstract**

The treatment of complexity is one of the main problems in the area of validation and verification of formally specified protocols. The problem can be tackled by using *heuristics*, *partial order simulation methods*, and *optimization strategies*. We present these mechanisms and describe the way they work. Their usefulness will be discussed by presenting the results of some experiments.

## 1 Introduction

In the areas of validation, verification and test generation some of the methods rely on a exhaustive simulation of system specifications. This methods have to deal with complex or even infinite state spaces of protocol specifications. The reasons for complexity are (1) the increasing power of modern protocol functions which have to be described by complex specifications, (2) characteristics of the chosen specification language which may facilitate some development and validation methods, but may not support test generation, and (3) missing information of the environment in which the protocol should work.

The characteristics mentioned in point 2 may concern the semantics of the specification language or its communication mechanisms. We will focus on the characteristics of SDL with its interleaving semantics. The problem of point 3 is that we are dealing with open systems. For an automatic simulation the behavior of the environment has to be modeled. The simple assumption that the environment is able to send and receive each valid signal at any time leads to an enormous amount of possible test cases. Complexity due to point 1 cannot be avoided. Our aim is to provide mechanisms for handling the complexity due to the points 2 and 3. We distinguish between three classes of reduction mechanisms called *heuristics*, *partial order simulation*, and *optimization strategies*.

*Heuristics* are based on assumptions about the behavior of the system to be tested, or its environment. They avoid the elaboration of system traces[1] which are not in accordance with the selected assumptions. *Partial order simulation* methods avoid complexity which is caused by an interleaving semantics of the specification language. They intend to limit the exploration of traces for concurrent executions[2]. *Optimization strategies* intend to reduce the possible behavior of the system environment. This can be done by using external information, e.g., specifications of surrounding services, or by analyzing the specification in order to generate optimal input data for the test environment.

In this paper we describe heuristics, partial order simulation methods and optimization strategies for SDL specifications. We implemented some of the described reduction strategies and present the results of some experiments.

[1]Throughout this paper a trace is meant to be a totally ordered sequence of events.

[2]A concurrent execution can be seen as a partially ordered set of events. All traces which do not violate the partial order describe the interleaved traces of the concurrent execution.

## 2 Foundations

Throughout this paper we will use some terms which may lead to ambiguities and misunderstandings. We introduce them briefly by assuming some basic knowledge about SDL.

### 2.1 Describing distributed systems by using SDL

A distributed system consists of several processes which work in parallel. In SDL a process is meant to be an extended input/output FSM. Extended FSM means that the FSM is able to store state information in local variables. Input/output FSM means that state transitions may be triggered by signals and that the FSM is able to send signals during a state transition. A state transition of an SDL process is described by a sequence of actions. Such an action may concern the consumption or sending of a signal, the setting or resetting of timers, or the manipulation of local variables.

Processes may communicate or synchronize with other processes. SDL processes communicate asynchronously by exchanging signals via communication links which are called channels and signalroutes. Signals sent, but not consumed are buffered in infinite FIFO queues. For this purpose each process has its own infinite signal queue. A process which wants to send a signal to a certain partner process has to put the signal into the signal queue of the partner.

A lot of complexity is introduced by the semantics of SDL that is based on *interleaving semantics*, i.e. the execution of an SDL specification is described by all interleaved traces of concurrently executable events. Another SDL specialty which often causes complexity problems is the treatment of time. SDL does not fix the duration of state transitions, but for the specification of time conditions a timer mechanism is provided. An SDL timer is an object which is associated with a process. It can be active or inactive. If it is active, after a predefined time it will put a timer signal into the signal queue of the process. The timer signal can be consumed and thus can trigger state transitions like any other signal. Complexity problems occur because without a duration for state transitions the duration of a timer set command can be treated only as comment, i.e., the timer may expire at any time. This treatment may lead to an enormous amount of possible system states and system traces.

### 2.2 Closed systems

Throughout this paper we assume that we deal with *closed SDL systems* only, i.e., systems which do not communicate with the system environment. This is no restriction, because it is always possible to model the behavior of the environment by means of special SDL processes which are able to send and receive all possible signals at any time. The advantage of this assumption is that we are able to treat the communication with the system environment in the same way as the communication among SDL processes. In the following processes which model the behavior of the environment are called *environmental processes*. The other processes are called *system processes*.

### 2.3 Behavior representation

The entire behavior of a (closed) SDL system can be treated as a labeled transition system (LTS), i.e., an infinite automaton. It is common practice to describe the behavior of such a system in form of a *behavior tree*. Figure 1 presents part of a behavior tree. The root *S0* of the tree describes the initial state and the leaves *S2* and *S3* denote final states. The other nodes describe states which are reached during the execution of the system. State transitions are represented by annotated edges. The annotations describe the events which lead to the corresponding state transition.

The meaning of the terms *event* and *state* may need some clarification. An *event* denotes an arbitrary atomic SDL event like input, output, or task. A *state* is meant to be a global system state which comprises the local states of the processes, the values of variables, and the contents of all queues. The local states of the processes are not only SDL states. Intermediate states between two atomic actions, i.e., within an SDL state transition, are considered also.

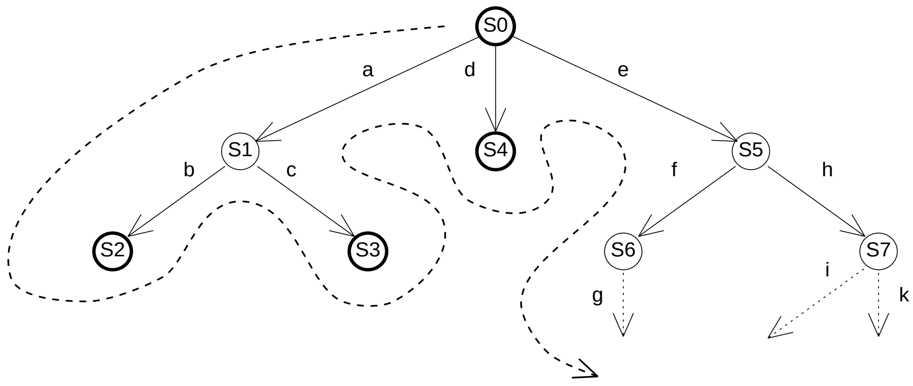

Figure 1: Behavior tree

In general, the behavior tree of an SDL system is not finite. One reason for this is that often an ongoing and never terminating behavior is required by the application area, e.g., a telephone system should not terminate. Another reason is the existence of infinite signal queues in SDL, which may lead to an infinite state space of the system.

By using an algorithm which explores for each state all its successor states recursively and an arbitrary SDL simulator which provides the functions *initialize*, *enabled-events*, and *execute-event* the behavior tree of an SDL specification can be generated easily. The generation of the behavior tree may be performed in a depth first order. Then the behavior tree is generated in the way shown by the dashed arrow in figure 1.

The behavior tree of an SDL specification grows exponentially. The degree of growth depends on the indeterminism which is specified in the system. In the case of closed systems it depends on various behavior possibilities of the environmental processes, indeterminism due to the interleaving semantics of SDL, and concurrency within the SDL system.

# 3 Reduction strategies

In this chapter we present present the investigated reduction strategies. We distinguish *heuristics*, *partial order simulation* methods, and *optimization strategies*. An overall view is given in section 3.4.

## 3.1 Heuristics

The first method for reducing the complexity of the behavior tree is the use of heuristics. Heuristics make assumptions about the systems that have to hold. They avoid the elaboration of system traces which are not in accordance with the assumptions. We distinguish between three kinds of heuristics: *limiting heuristics*, *filtering heuristics*, and *SDL specific heuristics.*

### 3.1.1 Limiting Heuristics

Limiting heuristics restrict the length of the traces to be elaborated, i.e., the behavior tree may be reduced to a finite depth. The character of limiting heuristics is shown schematically in figure 2 (a). We present four limiting heuristics: *maximal number of events*, *no double final state*, *no double state*, and *selection of states.*

**Maximal Number of events.** The heuristics *'maximal number of events'* defines a maximal number of events to be executed for each trace. If this upper bound is reached the further elaboration of this trace is stopped. The activation of this heuristics makes the (unlimited) depth search to a $k$-bounded depth search, i.e., the behavior tree is limited to the depth of $k$.

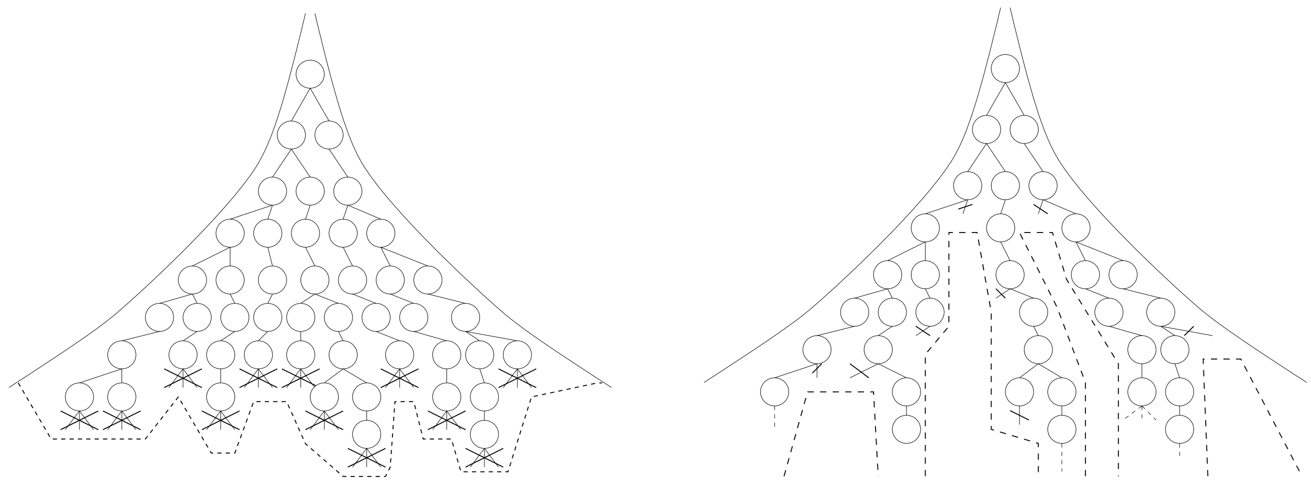
(a) Limiting heuristics (b) Filtering heuristics

Figure 2: Meaning of limiting and filtering heuristics

**No double final State.** The heuristics *'no double final state'* stops the investigation of a trace as soon as a final state is reached. But for many systems there are no states defined to be final states. In such cases the initial state of the system may be chosen to be a final state. The initial state of an SDL specification is the first stable state reached after having started the system, without performing any action of the system environment.

**No double State (Supertrace).** The idea of the heuristics *'no double state'* is to stop the exploration of a trace if a state is reached which already has been reached before. This can be done because the (sub-) behavior tree following the already reached state will be the same as the one following the first occurrence of the state. Since a system may have an infinite amount of states, it is a little bit tricky to implement this heuristics. Holzmann presents this heuristics in [6].

**Selection of states.** The heuristics *'selection of states'* selects a set of states of the specified system. We distinguish between a *positive* and a *negative* type of this heuristics. The *positive* type selects the states that may be reached during the simulation. If a state is entered which is not contained in the selection, the exploration of the actual trace is stopped. The *negative* type selects the states which shall not be taken into consideration. The exploration of a trace is stopped if a state is reached which is contained in the selection set.

### 3.1.2 Filtering Heuristics

Filtering heuristics reduce the branching factor at some states by suppressing the elaboration of some of its successors. Roughly spoken, filtering heuristics make a behavior tree thinner, but usually not finite. The character of filtering heuristics is depicted schematically in figure 2 (b). We present the filtering heuristics *strong reasonable environment*, *strong reasonable timers*, *first sent first consumed*, *maximal length of signal queues*, and *selection of signals.*

**Strong reasonable environment.** This heuristics makes an assumption about the behavior of the environmental processes. It assumes that after sending a stimulus to the system processes the environmental processes always waits until the system reaches a stable state. A *stable state* is meant to be a global system state, where the system is blocked until it receives a signal from the environment or until a timer expires. Roughly spoken, this heuristics avoids the overflow of signal buffers caused by the behavior of the system environment.

**Strong reasonable timers.** The idea behind the heuristics *'strong reasonable timers'* is that timers are usually used for exception handling. Due to the semantics of SDL it is not possible to simulate real time. Therefore a timer signal may always be sent out when a timer is active. In order to avoid this it is assumed that timers only expire if the system is in a stable state, i.e., timers are not allowed to expire while the system is busy.

**First sent first consumed.** The heuristics *'first sent first consumed'* defines an order upon SDL input events. During the simulation always the oldest sent signal is consumed first. Concurrently enabled SDL input events are not taken into consideration.

**Maximal length of signal queues.** The heuristics *'maximal length of the signal queue'* restricts the length of the infinite SDL signal queues. If a queue is full, then a sending event to this queue is not executable.[3] This heuristics avoids an infinite state space because of infinite signal queues. However, in [11] it is shown that under certain conditions the length of the input queues can be reduced without running the danger to loose concurrent executions.

**Selection of signals.** The heuristics *'selection of signals'* selects a set of signals of the specified system. We distinguish between a *positive* and a *negative* type of this heuristics. The *positive* type selects the signals which may be sent during the simulation. If the next event that should be performed is an output of a signal not contained in the selection the exploration of the actual trace is stopped. The *negative* type selects the signals which shall not be sent during the simulation. The exploration of a trace is stopped if the event which should be performed next is an output of a signal which is contained in the selection set.

### 3.1.3 SDL specific heuristics

The heuristics presented above may be applicable to arbitrary specification languages which are based on extended input/output FSMs which communicate asynchronously by using infinite signal buffers. In the following we present the heuristics *'indivisible SDL state transition'* and *'no null consumption'*. They are special filtering heuristics which make use of SDL specialties.

**Indivisible SDL state transitions.** Our behavior description for SDL systems (cf. Section 2.3) splits state transitions of SDL processes, i.e., transitions from one SDL state to another SDL state, into a sequence of atomic actions. This means we introduce intermediate states within the specified state transitions. Due to an interleaving where concurrency is modeled by interleaved traces within a behavior tree branching may also happen at the introduced intermediate states. The heuristics *'indivisible SDL state transitions'* suppresses this branching by executing all atomic SDL events belonging to an SDL state transition without interruption. The heuristics *'indivisible SDL state transitions'* is in line with the informal semantics of SDL provided by the language definition [8]. It only provides problems if in addition to the signal based communication a communication via global variables is used, i.e., *import*, *export*, and *view* constructs.

**No null consumptions.** The semantics of SDL provides the mechanism of null consumptions. This means that the first signal within the signal queue of an SDL process can be discarded if the signal cannot be consumed in the actual state of the process. It is obvious that such a mechanism facilitates the use of SDL since the specifier can concentrate on the relevant signals while other signals cause no deadlock situation. However, a null consumption should be an exception and not the usual case. Therefore the heuristics *'no null consumptions'* avoids the elaboration of traces which include null consumptions, i.e., the behavior tree will be limited to traces which do not include null consumptions.

[3] Another possibility to treat this situation is not to block the send event, but to discard the sent signal.

## 3.2 Partial order simulation

*Partial order simulation* methods can be seen as a special kind of filtering heuristics with a sound mathematical basis. They attempt to reduce complexity which is caused by an interleaving semantics of the specification language. SDL is based on interleaving semantics. This means concurrency is introduced by indeterminism, i.e., the execution of an SDL specification is described by all interleaved traces of concurrently executed events.

Related problems occurred in the area of protocol verification. They have been tackled by the development of partial order simulation methods [1, 9]. Partial order simulation methods attempt to limit the exploration of traces for concurrent executions. In the best case for each concurrent execution only one trace is generated.

Subsequently, we developed two algorithms which adapt the ideas of partial order simulation methods to the needs of SDL. The algorithms are called *independence prioritizing simulation* and *condition locking simulation*. We describe them informally. The details can be found in [11, 12].

### 3.2.1 Independence prioritizing simulation

During the simulation for each state the concurrently enabled events are identified. If such events never influence each others execution they are called to be *independent*. Independent events can be executed in arbitrary order, there is no influence in the resulting concurrent execution.

The independence prioritizing simulation algorithm takes advantage of this fact. During the simulation in each state it looks for an enabled event which is independent of all other enabled events. If such an event exists it is executed. The other enabled events are not treated as alternatives, i.e., as new branches in the behavior tree.

### 3.2.2 Condition locking simulation

The independence prioritizing simulation algorithm is not optimal, i.e., it may elaborate more then one trace for each concurrent execution. It only searches statically for independent events, but it does not consider actual dependencies. Furthermore only global independence and not the mutual dependencies between events are taken into account. The dependency relation is not a static relation. The execution of some events may have an influence on the dependency relationships between other events. The condition locking simulation algorithm always remembers the actual dependencies and therefore elaborates only one trace for each concurrent execution.

## 3.3 Optimization strategies

As mentioned before we assume that we deal with closed SDL systems (c.f. section 2.2). With respect to the environmental processes there are some optimization areas left to be explored. In the sequel we will focus on optimization strategies concerning (1) the behavior of the environmental processes, (2) the data flow (between environmental and system processes), and (3) the behavior of environmental processes compared with the service which should be provided by the system processes The points 1 and 2 are related to an analysis of the system processes before start of simulation. Point 3 is related to additional information.

### 3.3.1 Behavior of the environmental processes

Until now the environmental processes have been specified in a very simple way, i.e., they are able to receive and send any signal at any moment in time. In order to reduce the complexity of the simulation due to this kind of simple environmental processes we aim to avoid the sending of signals that lead to a null consumption on the side of the system processes. Therefore these processes should be analyzed with respect to this aspect. Although this may result in a more complex environmental process the complexity of simulation will be reduced.

#### 3.3.2 Data flow

During simulation the environmental processes send signals to the system processes. These signals may carry parameters, e.g., data packets or sequence numbers. Usually, there is a large range of valid values for these parameters and these values may influence the behavior of the system processes.

For example, after transmitting a data packet the response of a correct sequence number results in an acknowledgment of the transmission. The response of an incorrect sequence number results in a negative acknowledgment and the data packet is retransmitted. Many parameter values cause the same behavior. For example, there is a large set of incorrect sequence numbers and they all cause a retransmission of the data packet, i.e., the same behavior. During test generation often it is impossible to try all possible parameter values. This may lead to an enormous, sometimes even infinite, amount of possible traces, i.e., to an infinite branching of the behavior tree at some state nodes.

The idea is to concentrate on essential parameter values, i.e., we intend to select a small set of significant parameter values. For example, one parameter value which causes a positive acknowledgment and one which causes negative acknowledgment. All relevant system states of the system processes remain reachable. In other words, we try to build equivalence classes on parameter values with respect to the behavior of the system processes. For the simulation only one arbitrary value of each equivalence class will be used.

#### 3.3.3 Service like behavior of the environmental processes

The two optimization strategies discussed before rely on the analysis of the system specification. The third possibility needs additional information about the service the system should provide. According to the specification of this service the behavior of the test processes can be limited furthermore. The idea is that the environmental processes should only show behavior which conforms to the service specification, i.e., the behavior of a service user making use of the service in a correct manner. For example, if the service does not allow data transmission before a connection has been established the environmental process should not act like this either. Roughly speaking, the idea is to specify the environmental processes in such a manner that they show the mirrored behavior of the service specification. However, for such an optimization a complete service specification is needed.

### 3.4 Overall view

We presented several reduction strategies which can be used for handling the complexity of SDL specifications. The methods have been classified according to how they work. The classification is summarized in table 1.

## 4 Experiments

### 4.1 The SAMSTAG method

Our application area is the (semi-) automatic generation of test cases for formally specified protocols. Based on the SAMSTAG[4] method [2, 3, 10] a tool has been implemented. The generation of test cases can be looked at as a search problem within the state space of a specification. For each test case we search a system trace with specific properties which builds the basis for the test case description. The properties are mainly given by a test purpose specifying the objective of the test case and additional requirements which, for example, may concern start and end state of the test case. Therefore test generation methods have to deal with the complexity of state spaces of protocol specifications.

In SAMSTAG we implemented the heuristics *'first send first consumed'* (FSFC), *'strong reasonable timers'* (SRT), *'strong reasonable environment'* (SRE), *'no double final state'* (NDFS), *'no null consumptions'* (NNC), *'indivisible SDL state transitions'*, and the partial order simulation methods *independence*

[4]SAMSTAG is an abbreviation for *'Sdl And Msc bAsed Test cAse Generation'*.

| Name | Heuristics | | | Part. Ord. | Opt. |
|---|---|---|---|---|---|
| | Limit. | Filt. | SDL spec. | Simulation | Strat. |
| Maximal Number of Events | X | | | | |
| No Double Finite State | X | | | | |
| No Double State | X | | | | |
| Selection of States | X | | | | |
| Strong Reasonable Environment | | X | | | |
| Strong Reasonable Timers | | X | | | |
| Maximal Length of Signal Queues | | X | | | |
| First Sent First Consumed | | X | | | |
| Selection of Signals | | X | | | |
| Indivisible SDL Transitions | | | X | | |
| No Null Consumption | | | X | | |
| Independence Prioritizing | | | | X | |
| Condition Locking | | | | X | |
| Optimal Behavior of Test Processes | | | | | X |
| Optimal Data Flow | | | | | X |
| Service Like Behavior | | | | | X |

Table 1: Classification of all discussed methods

| 1 Sec | 1 Min | 1 hour | 1 day | 1 year | under heuristics |
|---|---|---|---|---|---|
| 8 | 13 | 18 | 22 | 29 | NO HEURISTICS |
| 9 | 16 | 22 | 27 | 36 | FSFC |
| 8 | 13 | 18 | 22 | 30 | SRT |
| 15 | 27 | 39 | 48 | 65 | SRE |
| 8 | 13 | 18 | 22 | 29 | NDFS |
| 8 | 13 | 18 | 22 | 29 | NNC |
| 9 | 16 | 22 | 27 | 36 | FSFC+SRT |
| 26 | 49 | 73 | 91 | 125 | FSFC+SRE |
| 25 | 47 | 69 | 86 | 117 | FSFC+SRT+SRE |
| 26 | 48 | 71 | 89 | 122 | FSFC+SRT+SRE+NDFS |
| 34 | 68 | 102 | 129 | 178 | FSFC+SRT+SRE+NDFS+NNC |

Table 2: Explored behavior tree by a given time and a given heuristics combination

*prioritizing simulation* (IPS) and *condition locking simulation* (CLS). The effects of these methods have been tested by means of the Inres protocol (cf. [4, 5]). The results of these experiments are described in this section.

## 4.2 Heuristics

We have tested the mentioned heuristics and several combinations of them. The results of these experiments are shown in table 2 and figure 3.[5] During the test runs we measured the average branching factor for the nodes of the behavior tree. Based on this factor some of the presented values are calculated. A logarithmic scale is used for the $y$ axis in figure 3. During the experiments the SDL specific heuristics *'indivisible SDL state transition'* always has been turned on.

However, table and figure show how deep the behavior tree of the Inres protocol can be explored in a given time by using one or a combination of several heuristics. It can be seen that the heuristics *'strong reasonable environment'* (SRE) is the most efficient one.

## 4.3 Partial order simulation methods

The partial order algorithms IPS and CLS also have been applied to the Inres protocol. We measured the speed of the algorithms, the size of the explored behavior tree, and the time for test generation. We

[5]In figure 3 some lines representing different heuristics combinations overlap and cannot be distinguished. This applies for the combinations NO HEURISTICS, NDFS and NNC, and for the combinations FSFC and FSFC+SRT.

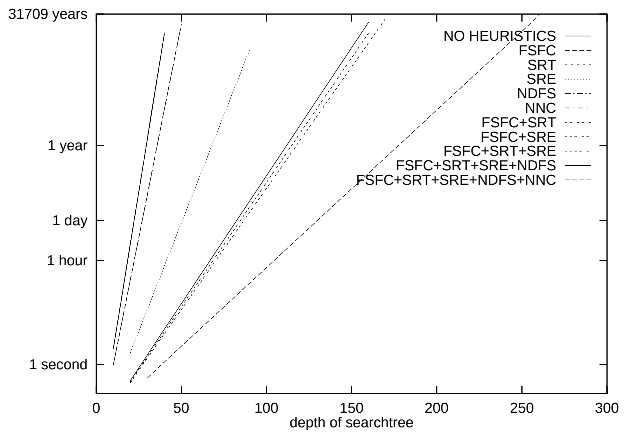

Figure 3: Needed time to explore the behavior tree

compared both algorithms with a normal *interleaving simulation* (ILS) which explores the behavior tree in the way described in Section 2.3.

### 4.3.1 Size of the generated behavior tree

In a second experiment the size of the generated behavior tree was examined. For a given depth we have counted the number of explored nodes. In a first test run all heuristics have been deactivated. The results are shown in figure 4 on the left. In a second test run all heuristics have been turned on. The results are presented in figure 4 on the right.

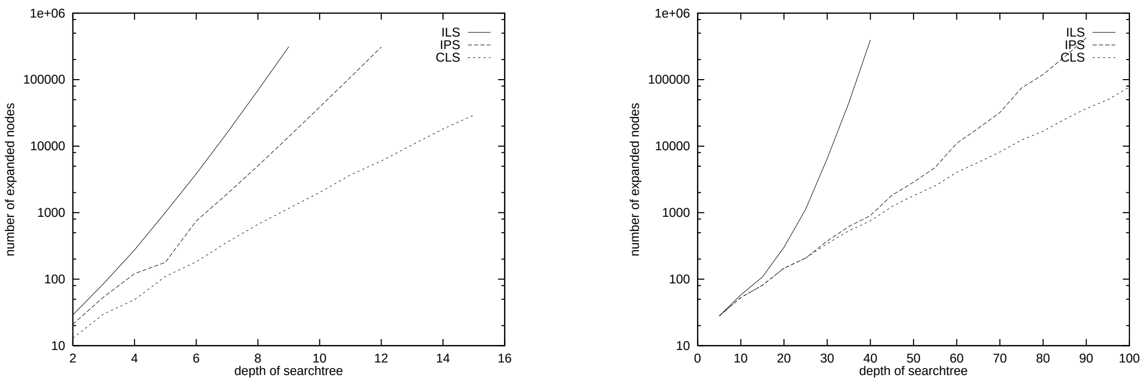

Figure 4: Size of the generated behavior tree without (left) and with (right) heuristics

### 4.3.2 Time for test case generation

In a third experiment we measured the time used by the different algorithms for generating test cases. For this purpose we integrated IPS and CLS into the SDL simulator of the SAMsTAG tool. The basis for the test generation was a test architecture for the Initiator instance of the Inres protocol according to the distributed test method (cf. [7]). The test purpose was to test a data transfer with acknowledgment after the first transmission of the data package. All heuristics have been turned off during the search for unique pass observables (cf. [3, 10]). The ILS algorithm was not able to generate the test case. The run was interrupted after 7 hours. At this time 10 million states have been explored. The IPS algorithm needs 8 minutes and 33 seconds for the test case generation, and the CLS algorithm needs only 3 minutes and 39 seconds. The advantage of partial order simulation methods is obvious.

### 4.4 Discussion

The experiments have shown the usefulness of heuristics and partial order simulation methods for test case generation. However, the effects achieved by heuristics and partial order methods may need some discussion.

The effects of partial order simulation methods are defined clearly. They reduce the number of interleaved traces for each concurrent execution of the SDL specification. No concurrent execution gets lost, but the explored behavior tree is reduced. Partial order simulation methods reduce complexity introduced by the interleaving semantics of SDL. They cannot reduce complexity due to the concurrency in the system. Figure 4 shows that the amount of time still grows exponentially with the depth of the explored behavior tree. Only the exponent is decreased.

The effects of heuristics often can not be described exactly. In general heuristics also reduce the size of the explored behavior tree, but it cannot be guaranteed that all relevant traces have been examined. For example, consider a situation where we search a trace which passes through a final state. By using the heuristics *'no double final state'* no solution will be found. However, we look at heuristics as a sort of tool box. The test specifier has to select a set of heuristics which is suitable for his problem.

## 5 Summary and outlook

We presented and discussed several methods which can be used for handling the complexity of SDL specifications. The methods have been classified according to how they work. Heuristics and partial order simulation methods have been implemented in the SaMsTaG tool. Their usefulness for test generation is shown by the results of some experiments. Together with Swiss PTT we started to apply SaMsTaG to the B-ISDN protocol SSCOP. During the test suite development we intend to implement mentioned optimization strategies.

## References

[1] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem (Preliminary Version).* PhD thesis, Universite de Liege, October 1994.

[2] J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts.* PhD thesis, University of Berne, Institute for Informatics and Applied Mathematics, February 1994.

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

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

[5] D. Hogrefe. OSI formal specification case study: The INRES protocol and service. Technical Report IAM-91-012, University of Berne, May 1991. Update May 1992.

[6] G. Holzman. *Design and Validation of Computer Protocols.* Prentice-Hall International, Inc., 1991.

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

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

[9] R. Langerak. True concurrency models for LOTOS. In D. Hogrefe and S. Leue, editors, *FORTE'94 - Tutorial Notes*, October 1994.

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

[11] D. Toggweiler. *Efficient Test Generation for distributed systems specified by automata.* PhD thesis, University of Berne, May 1995.

[12] D. Toggweiler, J. Grabowski, and D. Hogrefe. Partial order simulation of SDL specifications. In A. Sarma R. Braek, editor, *SDL'95 - with MSC in CASE*, North-Holland, September 1995.