Partial Order Simulation of SDL Specifications

Daniel Toggweiler, Jens Grabowski, and Dieter Hogrefe

University of Berne, Institute for Informatics, Neubrückstr. 10, CH-3012 Berne, Switzerland, {toggweil, grabowsk, hogrefe}@iam.unibe.ch

The need of efficient simulation methods for validation and verification of protocol specifications leads to the development of partial order simulation methods. Two new algorithms for the partial order simulation of SDL specifications are presented. Both algorithms have shown to be useful for the automatic generation of test cases. They are implemented in the test case generation tool SaMsTaG. The results of some experiments are discussed.

1. Introduction

Due to complexity the possibilities for validation and verification of communication protocols often are very restricted. A lot of complexity is introduced by the semantics of the specification language which is used to describe the behavior of the protocol functions. For example, SDL [10] is based on *interleaving semantics*. Concurrency is introduced by indeterminism, i.e., the execution of an SDL specification is described by all interleaved traces[1] of concurrently executed events.

Exploring all interleaved traces is not always necessary for verification. Traces which correspond to the same concurrent execution contain related information. Subsequently, *partial order simulation methods* exist for verification [1,4,6,9]. They attempt to limit the exploration of traces for concurrent executions. At best for each concurrent execution only one trace is generated.

We intend to improve the automatic generation of test cases for SDL specifications by using partial order simulation methods. Our test case generation method is SaMsTaG[2] [2,7]. SaMsTaG generates test cases by searching system traces with specific properties in the state space of an SDL specification. The most problematic point of SaMsTaG is the search procedure. Until now, due to the SDL interleaving semantics, SaMsTaG investigated all interleaved traces of the given specification within the search algorithm. But for test case generation in most cases it is sufficient to examine one trace only for each concurrent execution.

Consequently, we developed and implemented the *Independence Prioritizing Simulation* (IPS) and the *Condition Locking Simulation* (CLS). These new algorithms adapt the ideas of partial order simulation methods to the needs of SaMsTaG. They are presented in this paper. However, the benefits of partial order simulation methods for SDL

[1]Throughout this paper a trace is meant to be a totally ordered sequence of events.
[2]SaMsTaG is an abbreviation for *'Sdl And Msc baSed Test cAse Generation'*.

specifications are not restricted to SAMSTAG. Therefore IPS and CLS are described independently from SAMSTAG.

Throughout this paper we assume that we simulate *closed SDL systems*, 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.

The paper is organized in the following way: Section 2 introduces some fundamental notions. A small example is introduced in Section 3. Throughout the paper it will be used to explain the mechanism of partial order simulation methods. The IPS algorithm is described in Section 4. Section 5 presents the CLS algorithm. The results of some experiments are discussed in Section 6. In Section 7 summary and outlook are presented.

2. Foundations

It is common practice to describe the behavior of an SDL system in form of a *behavior tree*. Figure 2 (a) presents a part of a behavior tree. The root of the tree *S0* describes the initial state and the leaves *S2* and *S3* denote final states. The other nodes describe states which are reached during the simulation 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 *state* and *event* may need some clarification. A *state* is meant to be an SDL system state which comprises the local states of the processes, the values of variables, and the contents of all queues. An *event* denotes an arbitrary SDL event like *input*, *output*, or *task*. In principle it makes no difference to use complete (atomic) state transitions[3] instead of SDL events only.

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

However, the traces of an SDL system can be examined by using an arbitrary SDL simulator which provides the functions *initialize*, *enabled-events*, and *execute-event*. We describe these functions by using the following BNF notation:

$< FunctionName > ([< Parameter > \{, < Parameter >\}^*])[\rightarrow < ReturnValue >]$

(1) $initialize() \rightarrow InitialState$: initializes the SDL simulator and returns the start state of the simulation.

(2) $enabled\text{-}events(State) \rightarrow StackOfEnabledEvents$: returns for a given state a stack of all enabled events, i.e., events which can be executed next.

(3) $execute\text{-}event(State, Event) \rightarrow NextState$: takes *State* as actual state, executes *Event*, and returns the new state.

[3] An SDL state transition is meant to be a sequence of events which is performed by one process and which leads from one SDL state to the next SDL state.

```
1   declare
2      S     stack of states := (initialize());
3      TR    stack of events := ();
4      AL    stack of stack of events := ();
5      UpperBound  integer constant := external;
6
7   InterleavingSimulation()
8   {
9      push(AL, enabled-events(top(S)));
10     while(¬isempty(top(AL)) ∧ length(TR) < UpperBound)
11     {
12         e := top(top(AL));
13         push(S, execute-event(top(S), e));
14         push(TR, e);
15         push(AL, rest(pop(AL)));
16         call :  InterleavingSimulation();
17     }
18     pop(S); pop(TR); pop(AL);
19  }
```

Figure 1. Interleaving Simulation (ILS) algorithm

The function *enabled-events* returns a *stack* of events. A stack is a data structure which can be used to store elements of some type. It can be accessed and manipulated only by applying the functions *push*, *pop*, *top*, *rest*, *isempty*, and *length*.

(4) $push(Stack, Element)$: pushes $Element$ on the top of $Stack$.

(5) $pop(Stack) \rightarrow Element$: removes and returns the top element of $Stack$.

(6) $top(Stack) \rightarrow Element$: returns the element which was last pushed on $Stack$.

(7) $rest(Stack) \rightarrow Stack$: removes the top element of $Stack$ and returns the resulting stack.

(8) $isempty(Stack) \rightarrow BooleanValue$: returns the boolean value $true$ if $Stack$ is empty and $false$ if $Stack$ is not empty.

(9) $length(Stack) \rightarrow IntegerValue$: returns the number of elements which are actually stored in $Stack$.

Based on the functions (1) - (9) we define an algorithm which is able to examine the traces of an SDL system. The algorithm is called *interleaving simulation* (ILS) and is shown in Figure 1.

The expression $(initialize())$ in line 2 denotes a stack which only includes the initial state of the SDL system. The empty parentheses () in the lines 3 and 4 describe empty

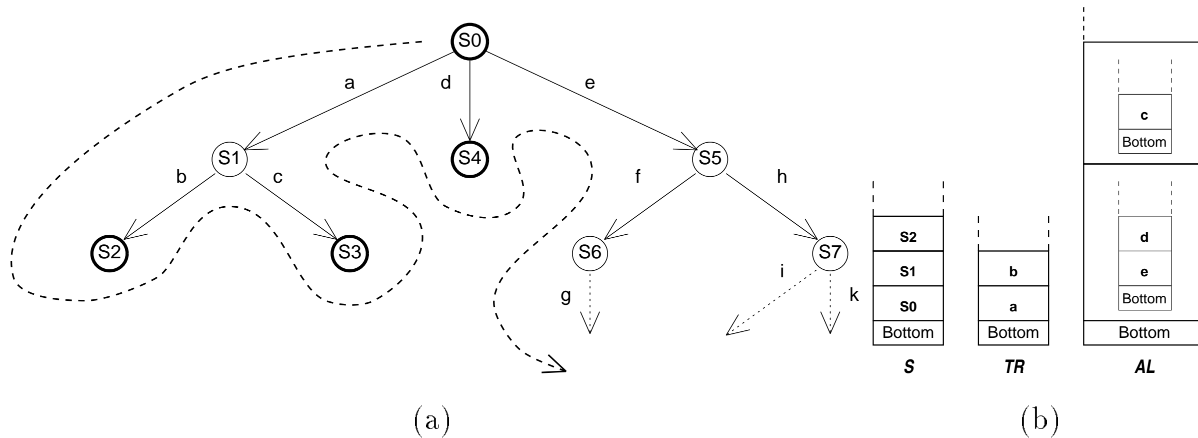


Figure 2. The interleaving simulation algorithm at work

stacks. The ILS algorithm makes use of the global data structures S, TR, AL, and $UpperBound$. The ILS algorithm always remembers the complete path, i.e., states and events, from the initial state to the actual state. The states are stored in the stack S and the events, i.e., the actual trace, are stored in TR. The alternatives are stored in AL. In case of infinite behavior a termination criterion is needed. We use a length restriction for the examined traces. The maximal length is given by the constant $UpperBound$.

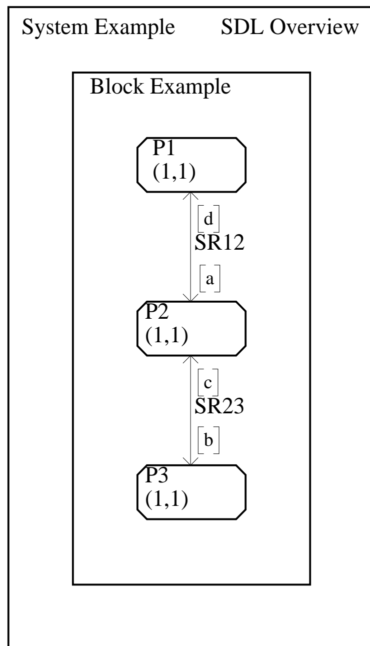
The way how the ILS algorithm explores the behavior of an SDL system is shown schematically in Figure 2. The dashed arrow in (a) indicates the way through the state space of the investigated system. (b) describes the contents of S, TR, and AL immediately after the execution of b, i.e., the actual state is $S2$.

3. Example

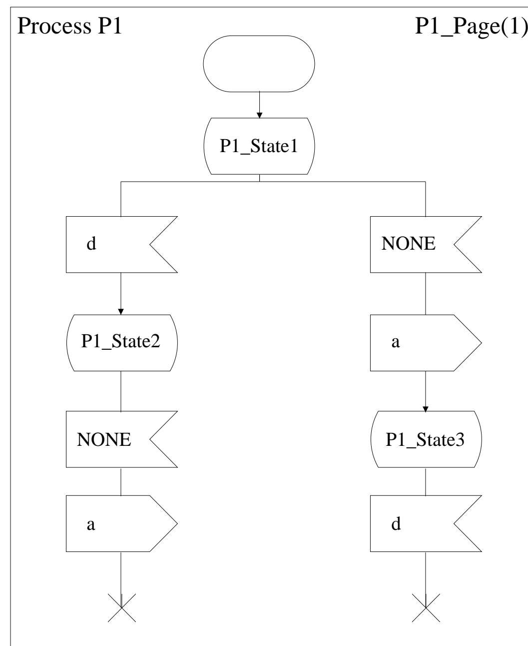
In Figure 3 a small SDL specification is presented. The system is called *Example*. It consists of the processes *P1*, *P2*, and *P3*. The processes do not communicate with the system environment, i.e., the system is closed. The system is finite, i.e., the system behavior ends in a global state where the local state of *P2* is *P2_State*, and the processes *P1* and *P3* are stopped. *P1*, *P2*, and *P3* exchange the signals a, b, c, and d.

Process *P1* may perform two events in arbitrary order. The events are the reception of d from *P2* and the sending of a to *P2*. Process *P3* has a similar behavior. It may perform the reception of b from *P2* and the sending of c to *P2* in arbitrary order. Process *P2* reacts on the reception of the signals a and c. On the input of a it gives b to *P3* and on the reception of c, signal d is sent to *P1*. A detailed analysis shows that some signals may cross each other. This applies for the signals a and d, and for the signals c and b.

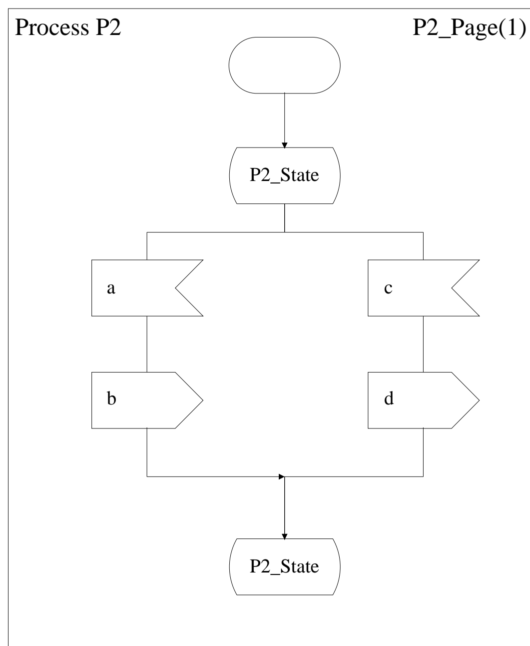
Applying the ILS algorithm will lead to an exploration of the behavior tree shown in Figure 4. The states are not inscribed with state names because we are interested in traces only. The edges are annotated with event descriptions. A *'!'* denotes an output event, e.g., the annotation *!a* describes the output of signal a, and a *'?'* describes an input, e.g., *?a* represents the input of signal a. The process which performs the send or receive event needs no special identification. Each signal can be sent, resp. received,



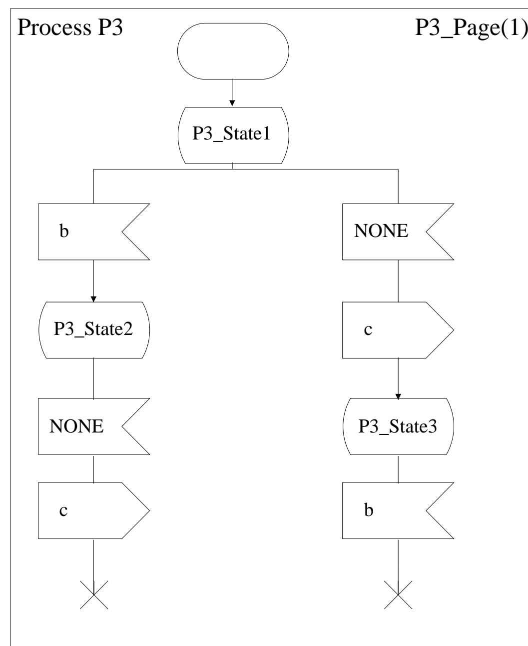
(a) System diagram



(b) Process definition of P1



(c) Process definition of P2



(d) Process definition of P3

Figure 3. SDL specification of the example system

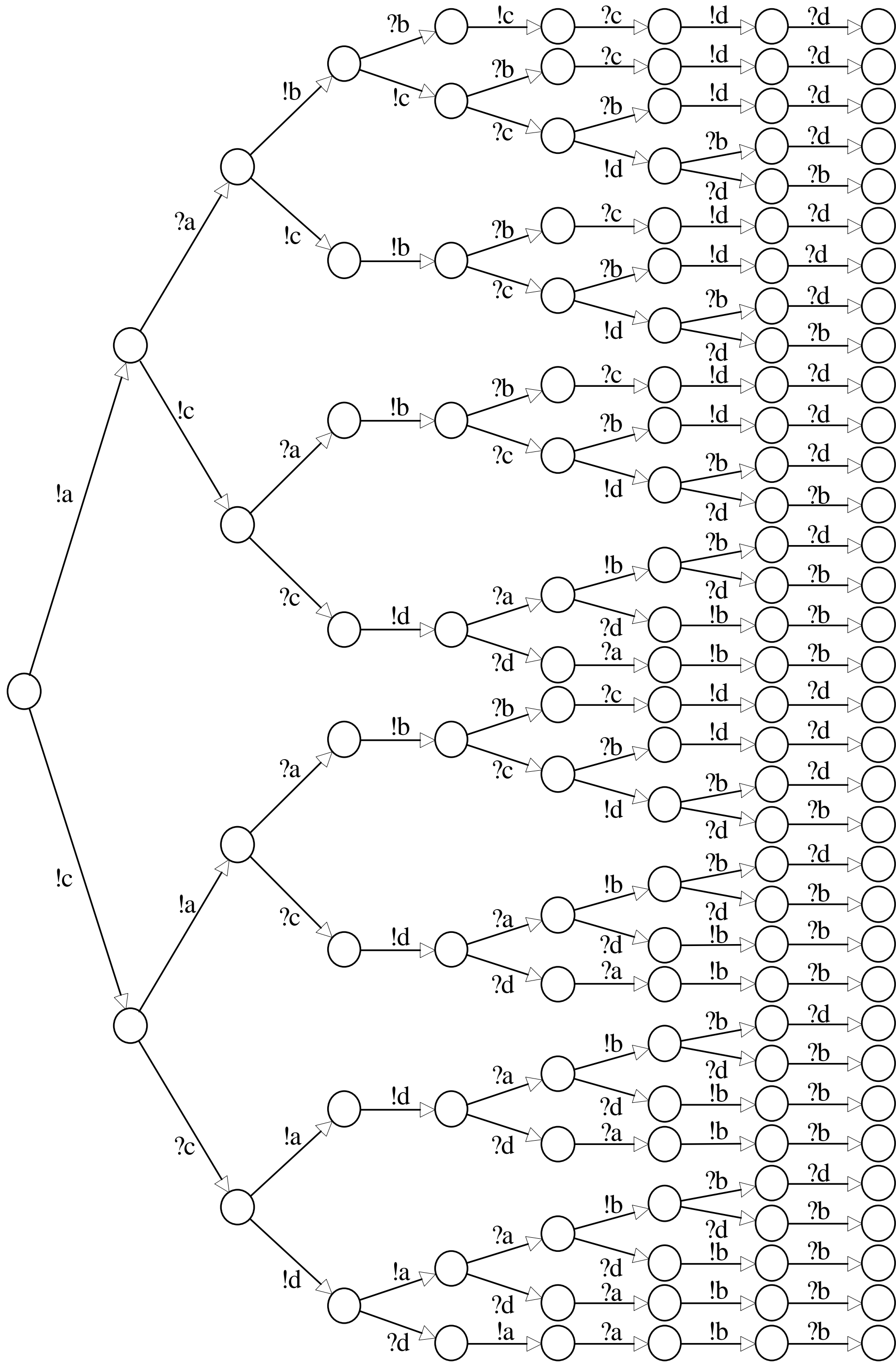


Figure 4. Applying the ILS algorithm to the example system

by one process only. The *input NONE* events in *P1* and *P3* are omitted because they describe no communication events. They specify the spontaneous sending of *a* and *c*. Since parallelism is described by all interleaved traces of concurrently executed events the behavior tree in Figure 4 looks complicated.

Various traces of Figure 4 belong to the same concurrent execution. Within a concur-

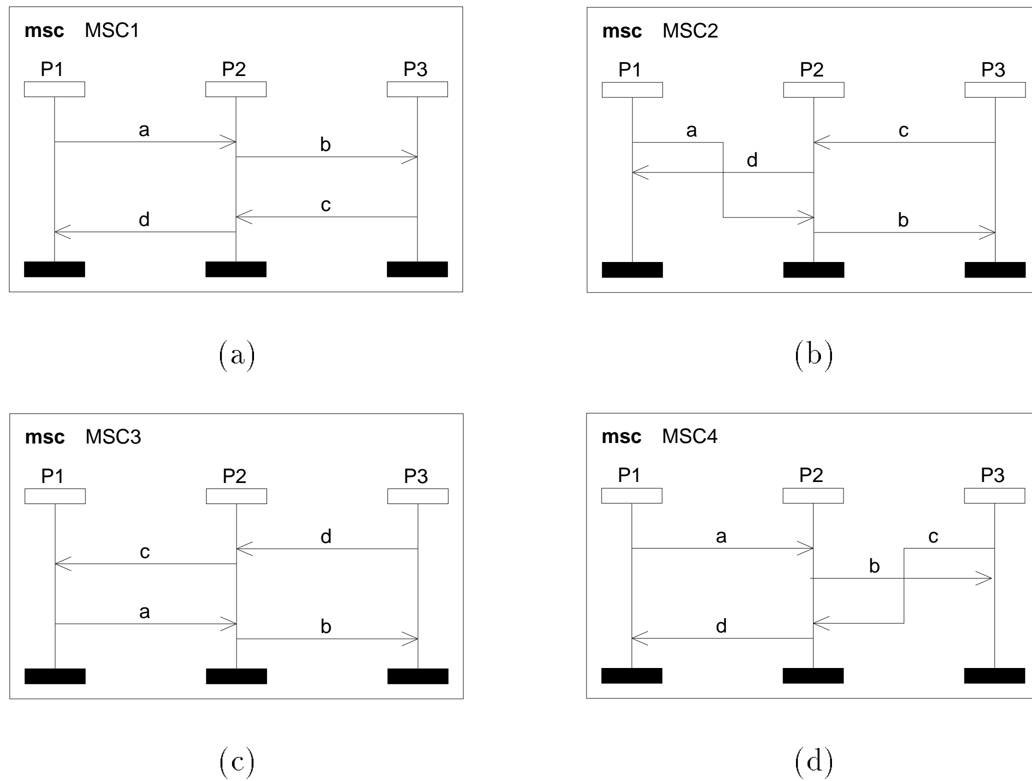


Figure 5. Concurrent executions of the example system

rent execution only the causal dependencies between events are considered which result in a partially ordered set of events in contrary to traces in which the events are ordered totally with respect to their execution time. The concurrent executions can be described in a compact and intuitive manner by using Message Sequence Charts (MSCs) [11]. Our SDL example (cf. Figure 3) has four concurrent executions. They are depicted in Figure 5. The valid traces belonging to an MSC are described by the traces which do not violate the partial order of the MSC. Our aim is to generate one trace only for each concurrent execution. By using the knowledge that the events of one process are totally ordered in time and that the order of events of different processes are mediated by messages an MSC representation of the concurrent execution can be constructed from each trace.

4. The Independence Prioritizing Simulation algorithm

During the simulation process the ILS algorithm identifies the concurrently enabled events for each reached state. Under certain conditions such events can be executed in arbitrary order. The idea of the *Independence Prioritizing Simulation* (IPS) algorithm is to take advantage of this fact. Independently from our work, a similar idea has been investigated by Holzmann and Peled [5]. But, their approach is adapted to the specific needs of verification and not to the needs of test case generation and the simulation of SDL systems.

However, at first we have to define the conditions which have to be checked during the simulation of the SDL system. We define three conditions which identify *dependent* events, i.e., events which cannot be executed in arbitrary order. For presenting compact condition formulas we need some definitions and notations.

For an arbitrary SDL system with a finite set of events[4] let

- O be the set of output events,
- I be the set of input events,
- ST be the set of spontaneous transitions (input NONE),
- T be the set of timeout events,
- P_e denote the process executing the event e,
- Q_e the input queue affected by the execution of event e,
- $initiates(e_i, e_j)$ a function which returns true, if e_i, e_j belong to the same SDL state transition and $e_i \in ST \cup T$, and
- $Dep(e_i, e_j)$ state that e_i and e_j are dependent.

Condition 1. Generally we assume that channels and signalroutes are undelayed. In this case output events, that do not originate from the same process, but do affect the same input queue, influence the concurrent execution. This is expressed formally by:

$$if\ e_i, e_j \in O \ \wedge \ P_{e_i} \neq P_{e_j} \ \wedge \ Q_{e_i} = Q_{e_j}\ then\ Dep(e_i, e_j)$$

Condition 2. Consider the case, where a signal can be consumed and a spontaneous transition or a timeout is possible. Then the SDL process has to make an indeterministic choice between the different events. The choice influences the concurrent execution. This is formally expressed by:

$$if\ P_{e_i} = P_{e_j} \wedge e_i, e_j \in I \cup T \cup ST \wedge (e_i \notin I \vee e_j \notin I)\ then\ Dep(e_i, e_j)$$

Condition 3. If an SDL state transition is initiated by an event e_d and another event e_j of this SDL state transition is dependent on another event e_k, then e_d must be dependent on e_k too. This is formally expressed by:

$$if\ Dep(e_j, e_k) \wedge initiates(e_d, e_j)\ then\ Dep(e_d, e_k)$$

The conditions 1 - 3 are based on the control and signal flow of an SDL system. Therefore they are only applicable to systems which do not share variables between processes, i.e., the use of the constructs view, export, import and remote procedure call is prohibited. A further restriction concerning the use of delayed channels has been mentioned already in the description of Condition 1.

It is obvious, that the existing conditions can be refined and precised. The advantage of the conditions 1 - 3 is, that they can be checked statically before the simulation starts. The dependencies between events do not change during run time. For describing the IPS algorithm a definition of the terms *independent* and *globally independent* is needed.

Definition of independent and globally independent. Let E be all events of an SDL system and e,$e_i \in E$.

- e and e_i are *independent* if none of the conditions 1-3 is valid.
- e is *globally independent* if $\forall e_i \in E \backslash \{e\}\ :\ e\ and\ e_i\ are\ independent.$

[4]Equal events in different SDL state transitions are meant to be different events.

```
 1   declare global
 2     S      stack of states := (initialize());
 3     TR     stack of events := ();
 4     AL     stack of stack of events := ();
 5     UpperBound  integer constant := external;
 6
 7   IndependencePrioritizingSimulation()
 8   {
 9     if (∃e ∈ enabled-events(top(S)) :  e is globally independent)
10     then push(AL, (e))
11     else push(AL, enabled-events(top(S)));
12     while(¬isempty(top(AL)) ∧ length(TR) < UpperBound)
13     {
14         e := top(top(AL));
15         push(S, execute-event(top(S), e));
16         push(TR, e);
17         push(AL, rest(pop(AL)));
18         call :  IndependencePrioritizingSimulation();
19     }
20     pop(S); pop(TR); pop(AL);
21   }
```

Figure 6. Independence Prioritizing Simulation (IPS) algorithm

Roughly spoken, the execution order of globally independent events which are enabled concurrently has no influence on the corresponding concurrent execution. Therefore we are allowed to assume an arbitrary execution order of these events without running the danger to loose any concurrent execution. This idea is realized in the *Independence Prioritizing Simulation* algorithm (IPS) which is shown in Figure 6. The IPS algorithm works in a similar manner as the ILS algorithm (cf. Figure 1). But, if there are globally independent events enabled in a state s, an arbitrary one is selected as the only transition which is executed from s (cf. lines 9-11 in Figure 6).

By applying the IPS algorithm to our example system the behavior tree (cf. Figure 7) is reduced from 34 to six paths. Each one of the events $?a$, $!b$, $?c$ and $!d$ is globally independent and the following six pairs of events are dependent: $(!a, !c)$, $(!c, !a)$, $(!a, ?b)$, $(?b, !a)$, $(?b, !c)$, and $(!c, ?b)$.

It is obvious, that the IPS algorithm is not applicable if the system behavior includes a loop of globally independent events. Furthermore the algorithm does not achieve any improvement if all events are dependent.

5. The Condition Locking Simulation algorithm

The application of the IPS algorithm leads to the exploration of six different traces although we can distinguish four concurrent executions only. In this section we present an

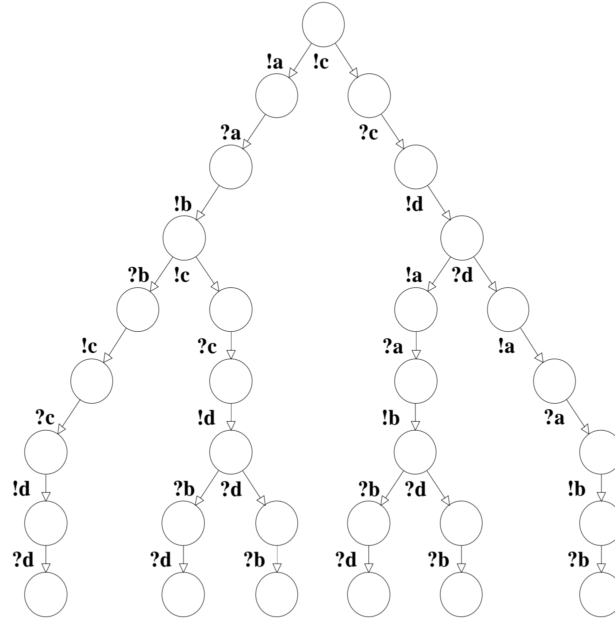


Figure 7. Applying the IPS algorithm to the example system

algorithm which explores one trace only for each concurrent execution.

The algorithm is explained by means of a finite system consisting of n events $e_1, \ldots, e_n$ which permits traces of a finite length only. Since the execution order of independent events may be neglected, a concurrent execution is defined by the execution order of the dependent events. We represent a concurrent execution uniquely by using an $n \times n$ matrix. Only some fields of the matrix are filled with boolean values. A field at location (i, j) is filled if the events e_i and e_j are dependent. The value is $true$ if e_i is executed before e_j. Otherwise it is $false$. The matrix is called *condition matrix* (CM) and the matrix fields $CM(1 \ldots n, 1 \ldots n)$ are called *conditions*. Now our problem of generating one trace only for each concurrent execution is reduced to the problem of computing each possible condition matrix exactly once.

Consider the case where an SDL system S with the initial state s_0 is in the actual state s. Let

- $s_1, \ldots, s_n$ be the possible successor states of s,
- $e_1, \ldots, e_n$ be the events leading from s to $s_1, \ldots, s_n$,
- Def_s be the set of conditions whose boolean values are defined by the simulation from the initial state s_0 to s (if s is the initial state $Def_s = \emptyset$),
- $Udef_s$ be the conditions whose values are not defined yet,
- $Tr_{e_1} := \{CM(1, n) \mid Dep(e_1, e_n) \wedge e_1 \; is \; executed \; before \; e_n\}$ is the set of conditions which evaluate to $true$ when e_1 is executed, and
- $Fa_{e_1} := \{CM(n, 1) \mid Dep(e_n, e_1) \wedge e_1 \; is \; executed \; before \; e_n\}$ is the set of conditions which evaluate to $false$ when e_1 is executed.

When e_1 is executed the system changes to the new state s_1. The sets Def_{s_1} and $Udef_{s_1}$ are calculated by $Def_{s_1} := Def_s \cup Tr_{e_1} \cup Fa_{e_1}$ and $Udef_{s_1} := Udef_s \backslash (Tr_{e_1} \cup Fa_{e_1})$.

The concept of the condition matrix and the calculation of the sets can be integrated into the IPS algorithm. All possible condition matrices will be generated at least once. The result will be the same as using the original IPS algorithm (cf. Section 4), i.e., certain condition matrices will be generated more than once. We describe the problem by using a small example.

Consider the situation where the system S is in the state s and the events e_1 and e_2 can be executed next. The events e_1 and e_2 are independent, but not globally independent. Therefore during simulation they are treated as alternatives. Both alternatives will be examined although e_1 and e_2 can be executed in arbitrary order without having any influence on the corresponding concurrent execution. This can be shown by calculating the sets defined above. Let s_{12} be the state which is reached if e_1 is executed before e_2 and let s_{21} be the state which is reached if e_2 is executed before s_1. Then:

$Def_{s_{12}} = (Def_s \cup Tr_{e_1} \cup Fa_{e_1}) \cup Tr_{e_2} \cup Fa_{e_2} = (Def_s \cup Tr_{e_2} \cup Fa_{e_2}) \cup Tr_{e_1} \cup Fa_{e_1} = Def_{s_{21}}$
and
$Udef_{s_{12}} = Udef_s \backslash (Tr_{e_1} \cup Fa_{e_1} \cup Tr_{e_2} \cup Fa_{e_2}) = Udef_s \backslash (Tr_{e_2} \cup Fa_{e_2} \cup Tr_{e_1} \cup Fa_{e_1}) = Udef_{s_{21}}$

The solution to this problem is to evaluate the simulation run starting in state s with event e_1 in the normal manner, and to *lock* the execution of e_1 when the alternative run starting in s with e_2 is examined until we can guarantee that an already computed condition matrix cannot be generated again. This can be guaranteed if a condition $CM(1,n) \in Tr_{e_1} \wedge Udef_s$ changes its value to $false$. Such a criterion can be checked during the simulation process.

The algorithm implementing the ideas of a condition matrix and event locking is called *Condition Locking Simulation* (CLS) algorithm. It creates each possible condition matrix exactly once. The proof for this and a formal description of the algorithm can be found in [8].

In the following the CLS algorithm is described informally by referring to the IPS algorithm presented in Figure 6 and explaining the changes to be made:

- Global data structures (lines 1-5)
  - In addition to the global data structures of IPS the CLS algorithm uses a condition matrix CM and a stack of stacks of locked events (LE).
- Evaluation of the enabled events (lines 9-11)
  - delete all events from the list of enabled events which are stored within the top element of LE.
- Forward steps (lines 14-18)
  - make a copy of the top element of LE and push it onto LE
  - delete all events from LE which are dependent to e
  - enter the changes caused by the execution of e into CM
  - after the forward step (line 18) push e onto the top element of LE
- Backward step (line 20)
  - undo the changes in CM caused by the execution of the event stored in the top element of TR
  - pop the top element from the LE stack

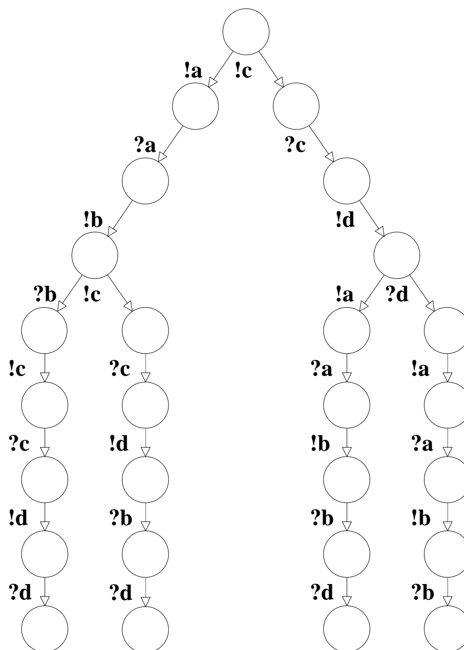


Figure 8. Applying the CLS algorithm to the example system

By applying the CLS algorithm to our example four traces are explored. The corresponding behavior tree is presented in Figure 8.

The CLS algorithm has been explained by means of a static and finite system which permits only traces of finite length. For the treatment of infinite SDL systems we generalized the CLS algorithm. The generalization has to cope with dynamic condition matrices. The details can be found in [8].

6. Experiments

The presented algorithms ILS, IPS, and CLS are implemented. We compared their capabilities by using them to simulate a test architecture of the Inres protocol [3]. Here we present the results of three experiments which have been carried out on a Sun Sparc 5 workstation.

Speed of the algorithms. We measured the number of states generated per second. The exact values of this experiment are shown in Figure 9. Astonishingly the ILS algorithm is slower then IPS and CLS. The reason for this is that, in general the ILS algorithm has to store more alternatively enabled events for each state.

Exploring a complete behavior tree up to a given depth. The results of this test are shown in Figure 10. The depth of the tree is described on the horizontal axis. The number of generated nodes is presented in logarithmic scale along the vertical axis. The experiments show that the partial order simulation algorithms allow to explore behavior trees up to a bigger depth than the interleaving simulation.

Generation of test cases. The algorithms have been implemented in the SaMsTaG test case generator [2,7]. We generated a test case for the Inres protocol. We used

Algorithm	Maximal Speed states per second	Minimal Speed states per second
ILS	1111	909
IPS	1111	1000
CLS	1111	1000

Figure 9. The speed of the algorithms

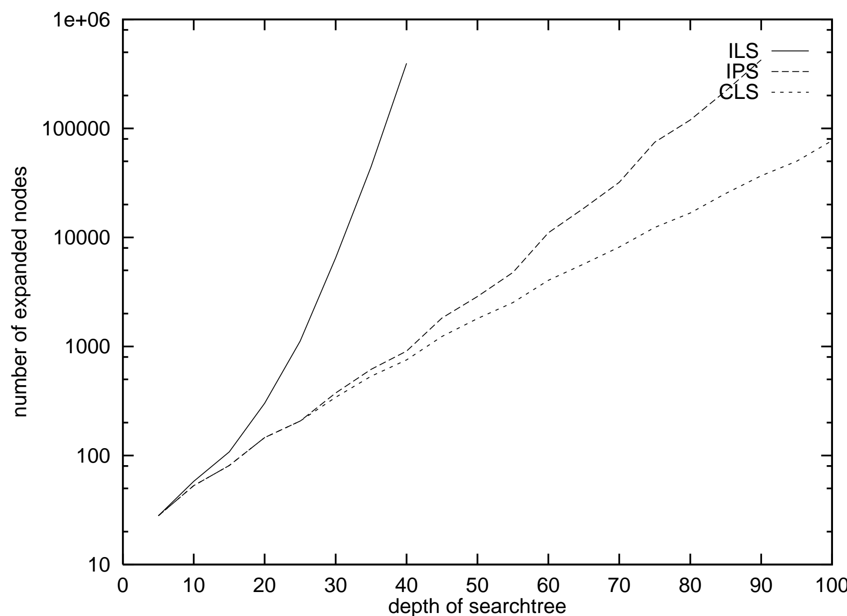


Figure 10. Exploring complete behavior trees

Algorithm	Consumed time
ILS	Interrupted after 7 hours, more than 10 mio states have been explored
IPS	8'33"
CLS	3'39"

Figure 11. Using ILS, IPS, and CLS for test case generation

a test purpose which checks the data transfer with an acknowledgement after the first transmission of the data package. The results of this experiment are shown in Figure 11. They emphazise the power of partial order simulation methods for test case generation.

7. Summary and outlook

We presented two algorithms which adapt the ideas of partial order simulation methods to SDL specifications. The algorithms are implemented and their power for the automatic

generation of test cases has been proven by some experiments.

However, the first version of the SAMSTAG tool uses interleaving simulation for test case generation. In order to reduce the complexity of the generation process we implemented additional heuristics like *reasonable environment* or *strong reasonable timers* [7]. These heuristics have proven to be useful for interleaving simulation. Consequently, we started to investigate whether these heuristics also improve the test case generation based on partial order simulation methods.

Acknowledgements

The presented work is funded partially by the KWF-Project No. 2555.1 *'Graphical Methods in the Test Process'*, the R & D project No. 299 *'Conformance Testing - A Tool for the Generation of Test Cases'* funded by Swiss PTT, and the SPP IF project *'The Automatic Generation of Test Purposes'*. The authors would like to thank Dr. E. Rudolph for proofreading and valuable comments.

REFERENCES

1. P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem. PhD thesis, Universite de Liege, Faculté des Sciences Appliquées, October 1994.
2. J. Grabowski, D. Hogrefe, R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. In: SDL'93 - Using Objects. North-Holland, October 1993.
3. 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.
4. G. Holzmann, P. Godefroid, D. Pirottin. Coverage Preserving Reduction Strategies for Reachability Analysis. In: Proceedings 12th IFIP WG 6.1 International Symposium on Protocol Specification Testing and Verification. North-Holland, June 1992.
5. G. Holzmann and D. Peled. An Improvement in Formal Verification. In: Proceedings of Seventh International Conference on Formal Description Techniques (FORTE'94) in Berne (Switzerland), October 1994.
6. R. Langerak. True Concurrency Models for LOTOS. In: FORTE'94 - Tutorial Notes, October 1994.
7. R. Nahm. Conformance Testing Based on Formal Description Techniques and Message Sequence Charts. PhD thesis, University of Berne, February 1994.
8. D. Toggweiler. Efficient Test Generation for Distributed Systems Specified by Automata. PhD thesis, University of Berne, May 1995.
9. P. Wolper, P. Godefroid. Partial-order Methods for Temporal Verification. In: CONCUR'93, 4th International Conference on Concurrency Theory, Lecture Notes in Computer Science, vol. 715, Springer-Verlag. August 1993.
10. Z.100 (1993), CCITT Specification and Description Language (SDL), ITU-T, June 1994.
11. Z.120 (1993), Message Sequence Chart (MSC), ITU-T, September 1994.