

2. Preliminaries

This chapter provides necessary background regarding SystemC, the IVL, symbolic execution, POR and basic definitions for state transition system, which can be used to model finite SystemC programs. Some parts of the SystemC and IVL description in Section 2.1 and Section 2.2 already appeared in [Le+13].

2.1. SystemC

SystemC is implemented as a C++ class library. It includes an event-driven simulation kernel and provides common building blocks to facilitate the development of embedded systems.

The structure of a SystemC design is described with ports and modules, whereas the behavior is described in processes which are triggered by events and communicate through channels. SystemC provides three types of processes with `SC_THREAD` being the most general type, i.e. the other two can be modeled by using `SC_THREAD`. A process gains the runnable status when one or more events of its sensitivity list have been notified. The simulation kernel selects one of the runnable processes and executes this process non-preemptively. The kernel receives the control back if the process has finished its execution or blocks itself by executing a context switch. A context switch is either one of the function calls `wait(event)`, `wait(time)`, `suspend(process)`. They will be briefly discussed in the following. Basically SystemC offers the following variants of `wait` and `notify` for event-based synchronization [GLD10; IEEE11]:

- `wait(event)` blocks the current process until the notification of the event.
- `notify(event)` performs an *immediate notification* of the event. Processes waiting on this event become immediately runnable in this *delta cycle*.
- `notify(event, delay)` performs a *timed notification* of the event. It is called a *delta notification* if the delay is zero. In this case the notification will be performed in the next *delta phase*, thus a process waiting for the event becomes runnable in the next *delta cycle*.
- `wait(delay)` blocks the current process for the specified amount of time units. This operation can be equivalently rewritten as the following block `{ sc_event e; notify(e, delay); wait(e); }`, where *e* is a unique event. Thus the `wait(delay)` variant will not be further considered in the following.

More informations on *immediate*-, *delta*- and *timed*-notifications will be presented in the next section which covers the simulation semantics of SystemC.

Additionally, the `suspend(process)` and `resume(process)` functions can be used for synchronization. The former immediately marks a process as suspended. A suspended process is not runnable. The `resume` function unmarks the process again. It is a form of delta notification, thus its effects are postponed until the next *delta* phase of the simulation. Suspend and resume are complementary to event-based synchronization. Thus a process can be suspended and waiting

for an event at the same time. In order to become runnable again, the process has to be resumed again and the event has to be notified.

2.1.1. Simulation Semantics

The execution of a SystemC program consists of two main steps: an *elaboration* phase is followed by a *simulation* phase. During *elaboration* modules are instantiated, ports and channels are bound and processes registered to the simulation kernel. Basically elaboration prepares the following simulation. It ends by a call to the `sc_start` function. An optional maximum simulation time can be specified. The simulation kernel of SystemC takes over and executes the registered processes. Basically simulation consists of five different phases which are executed one after another [IEE11].

1. *Initialization*: First the *update* phase as defined in step 3 is executed, but without proceeding to the subsequent *delta notify* phase. Then all registered processes, which have not been marked otherwise, will be made runnable. Finally the *delta notify* phase as defined in step 4 is carried out. In this case it will always proceed to the *evaluation* phase.
2. *Evaluation*: This phase can be considered the main phase of the simulation. While the set of runnable processes is not empty an arbitrary process will be selected and executed or resumed (in case the process had been interrupted). The order in which processes are executed is arbitrary but deterministic¹. Since process execution is not preemptive, a process will continue until it terminates, executes a wait statement or suspends itself. In either case the executed process will not be runnable. Immediate notifications can be issued during process execution to make other waiting process runnable in this evaluation phase. Once no more process is runnable, simulation proceeds to the *update* phase.
3. *Update*: Updates of channels are performed and removed. These updates have been requested during the evaluation phase or the elaboration phase, if the *update* phase is executed (once) as part of the initialization phase. The evaluation phase together with the update phase corresponds to a *delta cycle* of the simulation.
4. *Delta Notify*: Delta notifications are performed and removed. These have been issued in either one of the preceding phases. Processes sensitive on the notification are made runnable. If at least one runnable process exists at the end of this phase, or this phase has been called (once) from the *initialization* phase, simulation continues with step 2.
5. *Timed Notification*: If there are timed notifications, the simulation time is advanced to the earliest one of them. If the simulation exceeds the optionally specified maximum time, then the simulation is finished. Else all notifications at this time are performed and removed. Processes sensitive on these notifications are made runnable. If at least one runnable process exists at the end of this phase, simulation continues with step 2. Else the simulation is finished.

After simulation the remaining statements after `sc_start` will be executed. This phase is often denoted as *post-processing* or *cleanup*. Optionally `sc_start` can be called again, thus resuming the simulation. In this case the *initialization* phase will not be called. The simulation will directly continue with the *evaluation* phase. An overview of the different phases and transitions between them is shown in Figure 2.1.

¹If the same implementation of the simulation kernel is used to simulate the same SystemC program with the same inputs, then the process order shall remain the same.

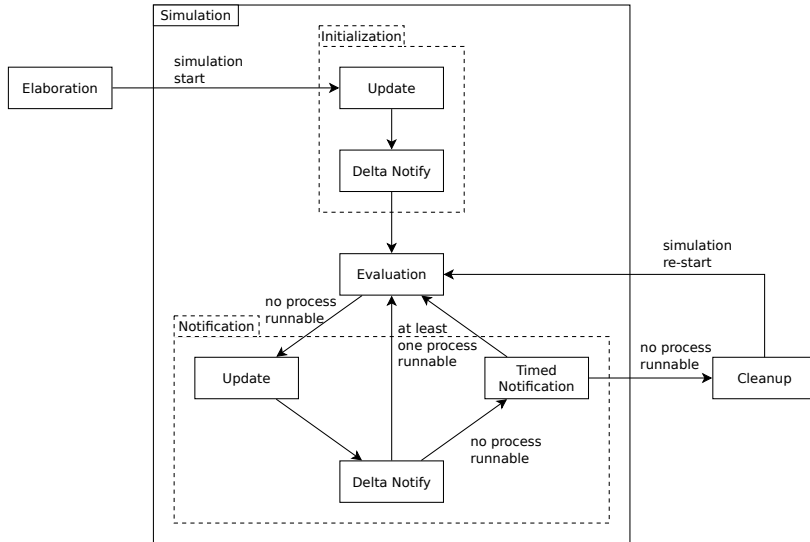


Figure 2.1.: Execution phases of a SystemC program. The *notification* phase is defined in this thesis as additional phase to group the *update*, *delta notify* and *timed notification* phases.

The non-determinism in the *evaluation* phase of the simulation, due to multiple process scheduling alternatives, is one of the reasons that give rise to the state explosion problem when it comes to the verification of SystemC programs. In order to assure that no failure is missed during simulation, it becomes necessary² to explore all relevant scheduling orders.

Remark. In the following the *update*, *delta notify* and *timed notification* phases will often be grouped as *notification* phase. Thus if simulation moves from the *evaluation* phase to the *update* phase, it will be said that the simulation is in the *notification* phase.

The interested reader is referred to [GD10; Gro02; Bla+09] or the IEEE standard [IEE11] for more details on SystemC.

2.2. SystemC Intermediate Verification Language

The SystemC Intermediate Verification Language (IVL) [Le+13] has been defined with the purpose of simplifying the verification process of SystemC programs, by separating it into two independent steps. The idea is that first a *front-end* converts a SystemC program into an IVL program, which is then verified by a separate *back-end*. The IVL has been designed to be compact and easily manageable but at the same time powerful enough to allow the translation of SystemC designs. A back-end should focus purely on the behavior of the considered SystemC program. This behavior is fully captured by the SystemC processes under the simulation semantics of the SystemC kernel. Therefore, a front-end should first perform the elaboration phase, i.e. determine the binding of ports and channels. Then it should extract and map the

²It becomes sufficient if all possible inputs are considered too, e.g. by employing *symbolic simulation*.

design behavior to the IVL. Separating the verification process of SystemC programs into two independent tasks makes both of them more manageable. In the following the structure and key components of the IVL are briefly discussed.

Based on the SystemC simulation semantics described in the previous section, three basic components of the SystemC kernel can be identified: *SC_THREAD*, *sc_event* and *channel update*. These are adopted to be *kernel primitives* of the IVL: *thread*, *event* and *update*, respectively. Associated to them are the following primitive functions in the IVL:

- *suspend* and *resume* to suspend and resume a thread, respectively
- *wait* and *notify* to wait for and notify an event (the notification can be either immediate or delayed depending on the function arguments, similar to the corresponding functions in SystemC)
- *request_update* to request an update to be performed during the update phase

These primitives form the backbone of the kernel. Other SystemC constructs such as *sc_signal*, *sc_mutex*, static sensitivity, etc. can be modeled using this backbone. The behavior of a *thread* or an *update* is defined by a function. Functions which are neither *threads* nor *updates* can also be declared. Every function possesses a body which is a list of statements. A statement is either an assignment, (conditional) goto statements or function call. Every structural control statement (if-then-else, while-do, switch-case, etc.) can be mapped to conditional goto statements (this task should also be performed by the front-end). Therefore, the representation of a function body as a list of statements is general and at the same time much more manageable for a back-end. As *data primitives* the IVL supports boolean and integer data types of C++ together with all arithmetic and logic operators. Furthermore, arrays and pointers of primitive types are also supported. For verification purpose, the IVL provides the *assert* and *assume* functions. Symbolic values of integer types and arrays are also supported.

2.2.1. Example

Basically an IVL program consists of a list of declarations. These include functions, threads, global variables and events. The execution of an IVL program starts by evaluating all global variable declarations. Then the (unique) *main* function will be executed. The *start* statement³ starts the actual simulation. An optional maximum simulation time can be passed as argument. If none or a negative value is passed, the simulation will not be time bounded. The simulation semantics directly correspond to those of SystemC, as described in Section 2.1.1.

Remark. The syntax for the *main* function and threads is slightly different to normal functions, since they neither take arguments nor return a result. Internally all of them are represented as functions though.

In the following an example SystemC and corresponding IVL program are presented. The main purpose of the example is to demonstrate some elements of the IVL. The example appeared in similar form in [Le+13]. It is presented here for convenience. For the sake of clarity, in this example and also in the following high level control structures are used in IVL programs instead of (conditional) gotos. Some slight syntactic adaptations have been performed to make the IVL easier to parse and read.

³One can think of it as a function too.

```

1  SC_MODULE(Module) {
2      sc_core::sc_event e;
3      uint x, a, b;
4
5      SC_CTOR(Module)
6          : x(rand()), a(0), b(0) {
7          SC_THREAD(A);
8          SC_THREAD(B);
9          SC_THREAD(C);
10     }
11
12     void A() {
13         if (x % 2)
14             a = 1;
15         else
16             a = 0;
17     }
18
19     void B() {
20         e.wait();
21         b = x / 2;
22     }
23
24     void C() {
25         e.notify();
26     }
27 };
28
29 int sc_main() {
30     Module m("top");
31     sc_start();
32     assert(2 * m.b + m.a == m.x);
33     return 0;
34 }

```

Listing 2.1: A SystemC example program

SystemC example Listing 2.1 shows a simple SystemC example. The design has one module and three `SC_THREADS` A, B and C. Thread A sets variable `a` to 0, if `x` is divisible by 2, and to 1 otherwise (line 13-16). Variable `x` is initialized with a random integer value on line 6 (i.e. it models an input). Thread B waits for the notification of event `e` and sets `b = x / 2` subsequently (line 20-21). Thread C performs an immediate notification of event `e` (line 25). If thread B is not already waiting for it, the notification is lost. After the simulation the value of variable `a` and `b` should be `x % 2` and `x / 2`, respectively. Thus the assertion (`2 * b + a == x`) is expected to hold (line 32). Nevertheless, there exist counter-examples, for example the scheduling sequence CAB leads to a violation of the assertion. The reason is that `b` has not been set correctly due to the lost notification.

IVL example Listing 2.2 shows the same example in IVL. As can be seen the SystemC module is "unpacked", i.e. variables, functions, and threads of the module are now global declarations. The calls to `wait` and `notify` are directly mapped to statements of the same name. Variable `x` is initialized with a symbolic integer value (line 2) and can have any value in the range of *unsigned int*. The statement `start` on line 24 starts the simulation.

```

1  event e;
2  uint x = ?(uint);
3  uint a = 0;
4  uint b = 0;
5
6  thread A {
7      if (x % 2)
8          a = 1;
9      else
10         a = 0;
11 }
12
13 thread B {
14     wait e;
15     b = x / 2;
16 }
17
18 thread C {
19     notify e;
20 }
21
22 main {
23     start;
24     assert (2 * b + a == x);
25 }

```

Listing 2.2: The example program of Listing 2.1 in IVL

2.3. Symbolic Execution

In principle symbolic execution [Kin76; CDE08] is similar to normal execution. A program is simulated by executing its statements one after another. The difference is that symbolic

execution allows to store and manipulate both symbolic and concrete values. A symbolic value can represent all or a subset of possible concrete value for each state part in the program. Thus it can be used to exhaustively check a single execution path for some or all input data. During execution a path condition pc is managed for every path. This is a Boolean expression that is initialized as $pc = \text{True}$. It represents constraints that the symbolic values have to satisfy for the corresponding path, thus effectively selecting the possible values a symbolic expression can evaluate to.

There are basically two different ways to extend a path condition: either by adding an assumption or executing a conditional goto. In order to add an assumption c , which itself is just a boolean expression, to the current path condition pc , e.g. by executing an *assume* statement, the formula $pc \wedge c$ will be checked for satisfiability by e.g. an SMT solver. If it is satisfiable, the path condition is update as $pc := pc \wedge c$. Otherwise the current execution path is considered *unfeasible* and will be terminated.

When a conditional goto with branch condition c is executed in a state s , which represents an execution path, an i.e. SMT solver is used to determine which of the branch condition and its negation is satisfiable with the current path condition. If both are satisfiable, which means both branches are *feasible*, then the execution path s is *forked* into two independent paths. One that will take the goto s_T and one that will not s_F . The path conditions of both paths are updated accordingly as $pc(s_T) := pc(s_T) \wedge c$ and $pc(s_F) := pc(s_F) \wedge \neg c$ respectively.

The symbolic execution effectively creates a tree of execution paths where the path condition represents the constraints under which a specific position in the program will be reached. In order to check whether an assertion of condition c is violated, the formula $pc \wedge \neg c$, where pc is the path condition under which the assertion is reachable, will be checked for satisfiability. The assertion is violated iff the formula is satisfiable.

The combination of symbolic execution with complete exploration of all possible process scheduling sequences enables exhaustive exploration of state spaces. The combined approach is called *symbolic simulation*. It is used by the symbolic simulator *SISSI* [Le+13] to discover assertion violations and other types of errors, such as memory access errors in SystemC IVL programs, or prove that none of them exists.

2.4. State Transition System

A state transition system (STS) is a finite state automaton describing all possible transitions of a system. In principle the definitions with regard to an STS follow the example of [FG05; KGG08; EP10].

Definition 1 (State Transition System (STS) or State Space)

A state transition system (STS), or state space, is a five tuple $A = (S, s_0, T, \Delta, s_e)$ where S is a finite set of states; $s_0 \in S$ is the initial state of the system; T denotes all possible transitions; $\Delta \subseteq S \times T \times S$ is the transition relation; $s_e \in S$ is a unique distinguished error state.

Let $A=(S, s_0, T, \Delta, s_e)$ be an STS. If $(s, t, s') \in \Delta$ then s' is the (unique) *successor* of s when executing transition t . It will be denoted as $s \xrightarrow{t} s'$. Sometimes the notation $s \xrightarrow{t}$ will be used to refer to s' . The transition t is said to be *enabled* in s . The set of enabled transitions in a state s is defined as $enabled(s) = \{t \mid (s, t, s') \in \Delta\}$. A state s with $enabled(s) = \emptyset$ is called *deadlock* or *terminal* state. The function $enabled(s)$ will sometimes be abbreviated as $en(s)$.

A (finite) sequence of transitions $w = t_1..t_n \in T^*$ is called a *trace*. A trace is executable from $s \in S$ iff there exists a sequence of states $s_1..s_{n+1}$ such that $s = s_1$ and $t_i \in \text{en}(s_i)$ and $s_i \xrightarrow{t_i} s_{i+1}$ for $i \in \{1..n\}$. The notation $s_1 \xrightarrow{w=t_1..t_n} s_{n+1}$ can be used to express the above situation. Thus the notation $s \xrightarrow{w} s'$ can be used for single or multiple transitions. Normally it is clear from the context which definition is used.

A state s' is said to be *reachable* from s iff there exists a trace w such that $s \xrightarrow{w} s'$, which can also be written as $s \xrightarrow{*} s'$ if the actual trace w is irrelevant. A state s is reachable in an STS iff $s_0 \xrightarrow{*} s$. Two transitions t and t' are *co-enabled* if they are both enabled in some reachable state s . Two traces $w_1 = t_1..t_n$ and $w_2 = a_1..a_m$ can be concatenated by the \cdot operation, thus $w_1 \cdot w_2 = t_1..t_n a_1..a_m$. Concatenation can also be used to prepend or append a single transition to a trace.

In the following the term *state space* will be used synonymously to refer to an STS. Sometimes it will even be referred to as (transition) *automaton*. The complete (global) state space is denoted as A_G to distinguish it from the reduced state space A_R that will be defined later during the presentation of state space reduction techniques. Instead of writing $s \in S$ it will often be simply said that s is in A or even $s \in A$, where A is an STS.

As a convention the existence of a (single) distinguished *error state* $s_e \in S$ is assumed. Once an error state is reached, the system will not leave it anymore, thus $\forall t \in T : (s_e, t, s_e) \in \Delta$. The predicate $\perp(s)$ will return True iff s is an error state, which by convention means that $s = s_e$. A transition that violates an assertion during execution will lead to an error state.

Remark. Due to symbolic execution, a single transition can lead to multiple successor states, when a conditional goto is executed where the branch condition and its negation are both satisfiable. For simplicity it will be assumed, and has been in the above description, that each transition has a single unique successor for each state. This is not a real limitation of the theoretical framework, since every non-deterministic automaton can be transformed into a deterministic one. Also this extension can be integrated quite naturally into the state space exploration algorithms that will be presented in this thesis, since all successor states are independent of each other. In practice, multiple successor states that arise due to symbolic execution can be handled quite efficiently. A state space exploration algorithm that explicitly handles multiple successors due to symbolic execution is provided in the Appendix. It will be specifically referred to during the presentation of state space exploration algorithms in Chapter 3.

2.4.1. Modeling IVL Programs as STS

As described in Section 2.2 the behavior of a SystemC program can be represented as IVL program. This section describes how finite IVL programs, i.e. the number of different states is finite and each transition runs for a finite number of steps, can be formally modeled as state transition systems.

A transition moves the system from one state to a subsequent state. In the IVL (and analogously SystemC) basically two different kinds of transitions can be identified, that change the state of the system in the simulation phase: *thread* and *notification* transitions.

Thread transitions change the state by executing a finite sequence of operations of a chosen thread followed by a context switch operation or termination of the same thread. Thus every thread can be separated into a list of transitions. The first transition begins with the first statement of the thread. All other transitions continue from a context switch statement that has interrupted the previous transition. All transitions either end with a context switch or the termination of the thread, which means that all statements of the process have been completely executed. Thus every thread T , that has not been fully executed, has a unique currently active

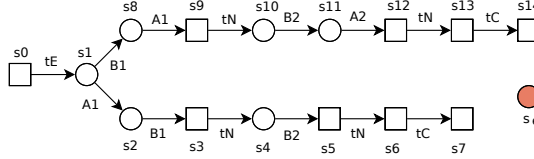


Figure 2.2.: Complete state space for the program in Listing 2.3

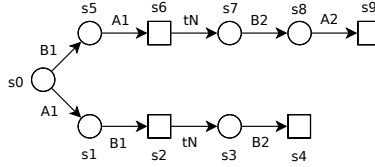


Figure 2.3.: Simplified state space for the program in Listing 2.3

transition in each state s , denoted as $next(s, T)$. This transition can either be enabled or disabled in s . Basically that is the transition that will be executed next when thread T is selected for execution in state s . A transition is enabled in a state s if it is the current active transition of a runnable thread.

For verification purposes, the SystemC IVL supports the *assume* and *assert* statements. Whenever an assertion violation is detected during the execution of a thread, the system will reach the designated error state s_e . The assume statement can be handled similarly, by introducing a designated terminal state, that is reached whenever the assumed condition is unsatisfiable.

A notification transition changes the state of the system by performing the *update*, *delta notify* and *timed notification* phases as necessary. The notification transition will be denoted as t_N in the following. A state s where $t_N \in en(s)$ is called a *notification* or *notify* state. According to the simulation semantics of SystemC, a notification transition will only be enabled if no thread transitions are runnable. Thus $t_N \in en(s)$ always implies that $en(s) = \{t_N\}$. All transitions between two notification states belong to the same *delta cycle*.

Thread and notification transitions are sufficient to model the simulation phase, which begins with the *start* statement. The execution of statements before and after the *start* statement can be modeled by introducing two additional distinguished transitions t_E and t_C respectively, similarly to the notification phase transition t_N . For the sake of simplicity these transitions will not be further considered in the following. The next section provides an example to illustrate the concepts of this section.

2.4.2. Example

Consider the simple IVL program in Listing 2.3. It consists of two threads A and B . Both of them consist of two transitions, separated by context switches, called A_1, A_2 and B_1, B_2 respec-


```

1  int a = 0;           8      }           15      notify eA;
2  int b = 0;           9      assert (b == 0); 16      }
3                      10     }           17
4  thread A {          11     }           18  main {
5      if (b > 0) {      12  thread B {      19      start;
6          wait eA;      13      b = 1;      20      }
7          wait_time 0;  14      wait_time 0;

```

Listing 2.3: Example to demonstrate the correspondence between IVL programs and state transition systems

tively. The transitions can⁴ execute the statements at lines:

$$\begin{aligned}
 A_1 &= \{5, 6, 9\} \\
 A_2 &= \{7, 9\} \\
 B_1 &= \{13, 14\} \\
 B_2 &= \{15\}
 \end{aligned}$$

The STS for this program is defined as $A=(S, s_0, T, \Delta, s_e)$ with:

$$\begin{aligned}
 T &= \{A_1, A_2, B_1, B_2, t_C, t_E, t_N\} \\
 \Delta &= \{(s_0, t_E, s_1), (s_1, A_1, s_2), (s_1, B_1, s_8), (s_2, B_1, s_3), (s_3, t_N, s_4), (s_4, B_2, s_5), (s_5, t_N, s_6), \\
 &\quad (s_6, t_C, s_7), (s_8, A_1, s_9), (s_9, t_N, s_{10}), (s_{10}, B_2, s_{11}), (s_{11}, A_2, s_{12}), (s_{12}, t_N, s_{13}), \\
 &\quad (s_{13}, t_C, s_{14})\} \\
 S &= \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_e\}
 \end{aligned}$$

A graphical representation of the STS is shown in Figure 2.2. Circles denote normal states, i.e. states where a transition can be selected, whereas squares denote states where the either one of the designated transitions $\{t_E, t_C, t_N\}$ is explored. In the following the t_E and t_C transitions will not be explicitly considered. Furthermore the last notification transition and unreachable error state will also normally be omitted. A graphical representation incorporating these simplifications is shown in Figure 2.3. Sometimes even the notification transitions in between will be omitted, since they can be unambiguously inferred given the original program. Error states will be shaded, as shown in Figure 2.2.

2.4.3. Remarks

This modeling requires not only that a SystemC program is finite but also that every transition considered in isolation will terminate. For example consider the simple IVL program in Listing 2.5. It would be invalid according to the above definition, since the transition of the thread *A* will not terminate, thus not reach a successor state. On the other hand a program with finite cyclic state space as shown in Listing 2.4 is valid, since every transition will eventually either finish execution or hit a context switch. While programs that loop without context switches are very interesting and challenging in the context of formal verification of generic programs, they are rather uncommon in the context of SystemC. For this reason such programs will not be further considered, though the methods presented in this thesis can be extended to support them.

⁴Whether the transition A_1 takes the branch in Line 5 depends on the state it is executed from.

```

1  thread A {
2    while (true) {
3      wait_time 0;
4    }
5  }
6
7  main {
8    start;
9  }

```

Listing 2.4: IVL program that can be modelled as STS.

```

1  thread A {
2    while (true) {
3    }
4  }
5
6  main {
7    start;
8  }

```

Listing 2.5: IVL program that cannot be modeled as STS due to non-terminating transitions.

Algorithm 1: Complete Stateful DFS

Input: Initial state

```

1   $H \leftarrow \text{Set}()$ 
2  explore(initialState)

3  procedure explore( $s$ ) is
4    if  $s \notin H$  then
5       $H.add(s)$ 
6      for  $t \in en(s)$  do
7         $n \leftarrow succ(s, t)$ 
8        explore( $n$ )

```

2.5. Basic Stateful Model Checking Algorithm

This section presents a basic stateful DFS algorithm that will explore the complete state space. It is shown in Algorithm 1. A set H is managed to store already visited states. The algorithm starts by calling *explore* with the initial state.

The *explore* procedure takes a state s as argument and will recursively explore all reachable states from s . First it checks whether s has already been explored. If not, s is added to the set of visited states H (line Line 5) and all enabled transitions in s are recursively explored one after another.

This basic algorithm performs a complete stateful exploration and thus suffers from the well-known state explosion problem. As has already been mentioned, POR and SSR will be applied in this thesis to alleviate the problem. The former explores only a subset of enabled transitions in Line 6. The latter uses a different equality predicate for the check $s \notin H$ in Line 4. Both reduction techniques can be applied together.

2.6. Partial Order Reduction

Partial Order Reduction (POR) is a widely used and particularly effective technique to combat the state explosion problem that arises in model checking of concurrent systems. The idea is to explore only a subset of the complete state space that is provably sufficient to verify the properties of interest.

It is based on the observation, that concurrent systems allow for the execution of many different transition interleavings, which yield the same resulting state. Thus it is sufficient to explore

Complete Symbolic Simulation of SystemC Models
Efficient Formal Verification of Finite Non-Terminating
Programs

Herd, V.

2016, XIX, 162 p. 26 illus., Softcover

ISBN: 978-3-658-12679-7