# Timing analysis of an SDL subset in Uppaal

Anders Hessel

April 16, 2002

**Abstract**

SDL is intended for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals.

Although the behaviour of an SDL system is clearly defined by the semantics of SDL, analysis of the real-time behavoiur is still hard to do for such a system. Timing and the arrival order of signals is of paramount significance for correct behaviour. So, it would be very difficult to do an exhaustive testing on an SDL system during run-time. This is partially because we cannot control the order of arrival of signals to a process, at least not for signals internal to the SDL system. No matter how many times we run a test suite, it is possible that signals always come in the same order at a specific state of the system.

Happily, there are verification tools for analysis of models of real-time systems e.g., Uppaal. The basis of the Uppaal model is the notion of timed automata developed by Alur and Dill as an extension of classical finite–state automata with clock variables. If we could transform an SDL system into a network of timed automata while conserving its behaviour we would be able to let Uppaal do the verification.

In this report we describe and implement a translation from an SDL (Specification and Description Language) syntax to a language (xta) used in the Uppaal tool. We show how it is possible to simulate implicit and explicit channels, queues, timers, dynamic process creation and process execution. Apart from timers we can also simulate worst and best execution time per action statement in the process.

## Acknowledgements

This master thesis projects started at ISD Datasystem AB [ISD] in March 1998. I want to thank ISD and its CEO Sven-Eric Berg for this opportunity. I also want to thank, as my supervisor at ISD, Jörgen Carlsson, for discussions on SDL sematics.

After ten weeks, sponsored by ISD, the work continued at my spare time. Around year 2000 a new Uppaal format was developed by the Uppaal team, namely the .xta format. This made it possible for me to express my translation output in templates. The project was now expanded to include type based processes and type based blocks.

In the SDL task there was first possible to express "target code" where the target code wasn't parsed at all. Now, with the new .xta I wanted to build a more controlled conversation and decided to fully parse all "target code" that also included some type checking. During this period I had professor Wang Yi as my supervisor, accompanied by the Uppaal group at Uppsala University. I like to specifically thank Paul Petterson, Wang Yi, Johan Bengtsson, and Fredrik Larsson for this time.

At late September 2001, professor Bengt Jonsson at Uppsala University became my supervisor and sponsor for the last coding and the writing of this report. I own him a lot of thanks for helping me with the report, both structurally and grammatically.

At last, I like to thank Madeleine for her patients during the years.

Anders Hessel, April 2002

# Contents

## List of Figures

# 1 Introduction

## 1.1 Overview and objectives

The objectives of this work is to describe, and implement, a translation from an SDL [Z.100] syntax to a language (xta) for the UPPAAL [LPY97] tool.

SDL is intended for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals.

Although SDL is widely used in the telecommunications field, it is also now being applied to a diverse number of other areas ranging over aircraft, train control, medical and packaging systems [SDL02].

Although the behaviour of an SDL system is clearly defined by the semantics of SDL, analysis of the real-time behavoiur is still hard to do for such a system. Timing is of paramount significance for correct behaviour. One of the problems is that the arrival order of signals to a queue matters for the behaviour. So, it would be very difficult to do an exhaustive testing on an SDL system during run-time. This is partially because we cannot control the order of arrival of signals to a process, at least not for signals internal to the SDL system. No matter how many times we test/simulate, it is possible that they always come in the same order at a specific state of the system.

Happily, there are verification tools for analysis of models of real-time systems e.g., UPPAAL. The basis of the UPPAAL model is the notion of timed automata [AD94] developed by Alur and Dill as an extension of classical finite–state automata with clock variables. If we could transform an SDL system into a network of timed automaton while preserving its behaviour we would be able to let UPPAAL do the verification.

The language used in our study is a variant of a subset of SDL. To separate SDL from our language we call our language $\mathcal{SDL}_{xta}$, (see Appendix B and C for the definition). The syntax of $\mathcal{SDL}_{xta}$ is based on SDL version 1992 (SDL-92). It includes most of the features from SDL-88, but the data types are taken from UPPAAL's .xta format. We have also added a syntax for adding best and worst case execution time (BCET/WCET) for each action in an SDL process. We give an overview of the differences in Appendix A, *Differences between SDL-92 and $\mathcal{SDL}_{xta}$*.

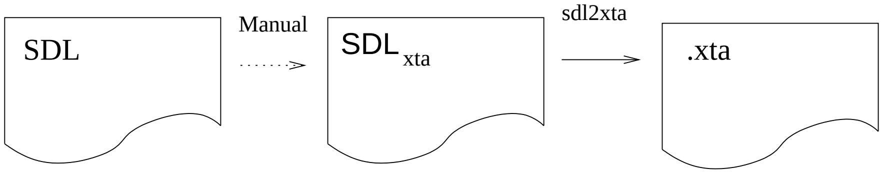


Figure 1: The path from SDL-92 to UPPAAL's .xta

## 1.2 Conversion problems

There are several problems that we have to solve or work around. Data types in SDL are of less interest and we don't include them in our language, $\mathcal{SDL}_{xta}$. Examples of problems we deal with in the $\mathcal{SDL}_{xta}$ language are simulations of: implicit and explicit channels, queues, timers, dynamic process creation and process execution. Apart from timers we can also simulate worst and best execution time per action statement in the process.

In SDL, it is possible to specify systems at many nested levels of abstraction. At the other hand, in UPPAAL, models are flat, i.e., there is no hierarchy other than the process level and the global level. So, apart from the local variables we had to flatten every context in $\mathcal{SDL}_{xta}$.

In $\mathcal{SDL}_{xta}$ a block or process can be specified as a type. That type can be used many times in its scope. Type instantiation has been one of the harder problems so solve, even if this matter is not specific to conversion into UPPAAL.

$\mathcal{SDL}_{xta}$ uses asynchronous signals that are queued at input to the receiving process. This is very different from the way timed automata work. When two timed automata synchronize, they simultaneously make a transition (rendez-vous). In SDL, when a process sends a message there is no guarantee that the receiving process is ready to receive that signal.

We have had a difficult task to implement the behaviour of the SDL queue, because it has a complex behaviour. A process can save a set of signals when it is in a specific state. Signals that are not in the save set and for which there are no transitions, are discarded.

When an SDL process sends out a signal without specifying the destination by process id, then the destination is determined by combining several hints in the output in conjunction with the system structure specification. The problem of signal delivery is solved in the conversion, and it may result in more than one receiving process for an output. This is called implict signal addressing and is one of the major problems for which we propose a solution in this report.

We also attack the problem of dynamic creation of processes. We cannot allow an unbounded number of processes but we can simulate dynamic creation up to a maximum. One problem that arises from this is that the process identity numbering makes the search space for UPPAAL infinite.

We have to interpret all flow control in the process including, the state machine itself, decisions and joins in the transitions.

We simulate a system where all processes are concurrent. We do not take in account whether two processes runs in the same block or not. In our model we don't have any delay on signal delivery at all. The only time that we take in account is the processes' timers, and the best/worst execution time of actions, when such time is supplied by the user.

As UPPAAL cannot compare a clock value with a variable, we have done our timers so they count discrete time units and thereby we can make a delay until a certain amount of time has passed.

## 1.3 Organization

The remainder of the document is organized in the following way:

- In Section 2, *SDL*, we give an introduction to SDL
- In Section 3, UPPAAL, we give an introduction to UPPAAL.

- In Section 4, *SDL run-time system building*, we present the architecture of, and the abstract ideas behind, the UPPAAL .xta code, that we generate. We also discuss signal analysis.

- In Section 5, *Process conversion*, we present the conversion of an SDL process. The communication between the queue and the process is explained. We also explain each SDL process action transition construct.

- In Section 6, *Brief example*, we give a first example.

- In Section 7, *Conclusion*, we discuss our achievements.

- In Appendix A, *Differences between SDL-92 and $\mathcal{SDL}_{xta}$*, we give an overview of the differences between SDL-92 and $\mathcal{SDL}_{xta}$.

- In Appendix B, *BNF of $\mathcal{SDL}_{xta}$*, we present a Backus Nauer Form (BNF) for $\mathcal{SDL}_{xta}$except the UPPAAL .xta part.

- In Appendix C, *Rules related to* UPPAAL *used in $\mathcal{SDL}_{xta}$*, we present a Backus Nauer Form (BNF) the UPPAAL .xta part in $\mathcal{SDL}_{xta}$.

- In Appendix D, *Usage of the conversion program*, we describe how to download and use the conversion program, `sdl2xta` described in this document.

- In Appendix E, *The signal analysis algorithm*, we outline the algorithm how possible receivers for an SDL output is found.

# 2 SDL

## 2.1 Definition

Specification and description language (SDL)[1] is an object-oriented, formal language defined by The International Telecommunications Union Telecommunications Standardization Sector (ITU-T) (formerly Comitè Consultatif International Telegraphique et Telephonique [CCITT]) as recommendation Z.100 [Z.100]. The language is intended for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals.

## 2.2 History

The development of SDL started in 1972. A 15-member study group within CCITT representing several countries and large telecom companies like Bellcore, Ericsson, and Motorola began research on a standard specification language for the telecommunications industry. The first version of the language was issued in 1976, followed by new versions in 1980, 1984, 1988, 1992, and 1996. The latest versions expanded the language considerably specifically the object oriented additions. Today SDL faces an itegration with OMG's *Unified Modeling Language* (UML) [SDL02].

## 2.3 Behaviour overview

An SDL system is defined by the behaviour of the processes it consists of and how they are interconnected. By ignoring many details we can describe a SDL system at run-time as **process** instances that communicate by sending **signals** to each other and/or to the environment (see Figure 2).

Each process can be viewed as a finite state machine [Holtz91] [Brau84], acting on input. Depending on its next input, the process performs a transition, that may include many actions, and finally moves to a new (possibly the same) state. Its next state may be determined by decisions in the actions. In SDL, a state is a location where only input from the queue can trigger a transition.

Communication in an SDL system is asynchronous. Figure 3 shows how transmitted signals are put in the queue of the receiving process instance. In Section 5.3 we will take a closer look at how a process instance interacts with its queue.

[1]Proposals have been made to re-assign the acronym to *System Design Languages [LNCS2078]*

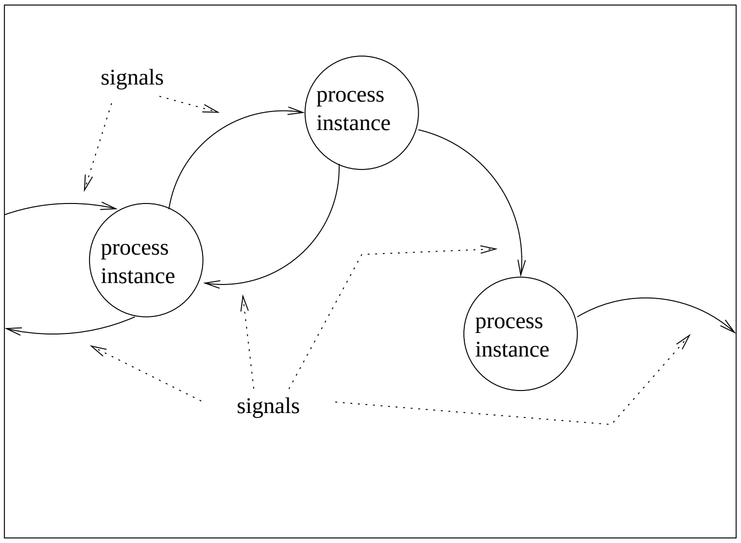


Figure 2: System behaviour of SDL.

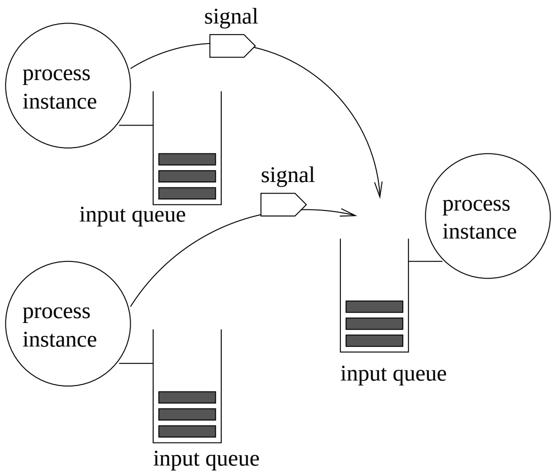


Figure 3: Process communication.

## 2.4 Structure

Structure is essential in SDL. At the top of the hierarchy of an SDL specification there is a **system**. A system contains **blocks**, futher a block contains **processes** or it could also be subdivided (or partitioned) into sub-blocks. We will call non partitioned blocks leaf-blocks. If a block is partitioned it contains only one **substructure**. A substructure contains blocks in the same manner as a system. We will not consider lower levels than the process level in this document.

We can thus identify the following hierarchical levels:

- system
- partitioned block
- ...
- partitioned block
- leaf-block
- process

**System level** A **system** is the highest level building block in the hierarchy. A system consists of blocks connected by **channels**, a block can also be connected with the environment, see Figure 4. As the system is the higest level, connecting to the environment means connecting to an environment outside the specification. This could be I/O from a human user as well as from another system. When connecting to the environment, the system is considered as open. In $\mathcal{SDL}_{xta}$ we will not allow an open system. In Section **??** we will address this issue more. A BNF for **system** in $\mathcal{SDL}_{xta}$ is defined in Section B.6.

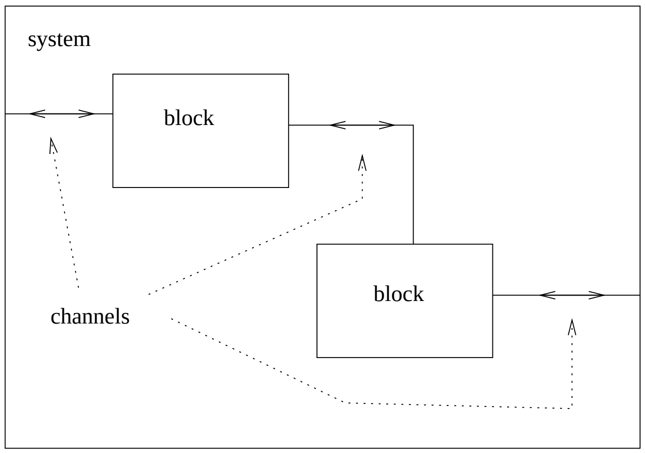


Figure 4: System with block.

**Partitioned block level** A partitioned block is a block containing a block substructure that itself contains blocks, just like a system. A partitioned block is shown in Figure 5. In a block substructure, channels can connect two blocks or connect a block to an external channel. In a block substructure, an internal channel, with one endpoint connected to the environment, can be connected to one external channel. The $\mathcal{SDL}_{xta}$ block substructure is defined in Section B.8.

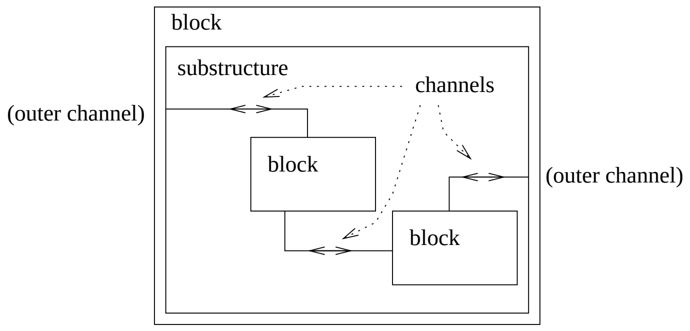


Figure 5: Block with substructure.

**Leaf-block level** In Figure 6, we show a **block** at leaf level. We will refer to this type of block as a leaf-block because is it a leaf in a hierarchy. A leaf-block contains processes. A process can be declared to have multiple instances. Therefore we refer to each process declaration as a process set. The communication channels in a leaf-block are called **signal routes**. A signal route can go from one process set to another process set or from a process set to the environment. A signal route that goes to the environment in a leaf-block is connected with a channel at the outside of the block. A BNF for **block** in $\mathcal{SDL}_{xta}$ is defined in Section B.7.

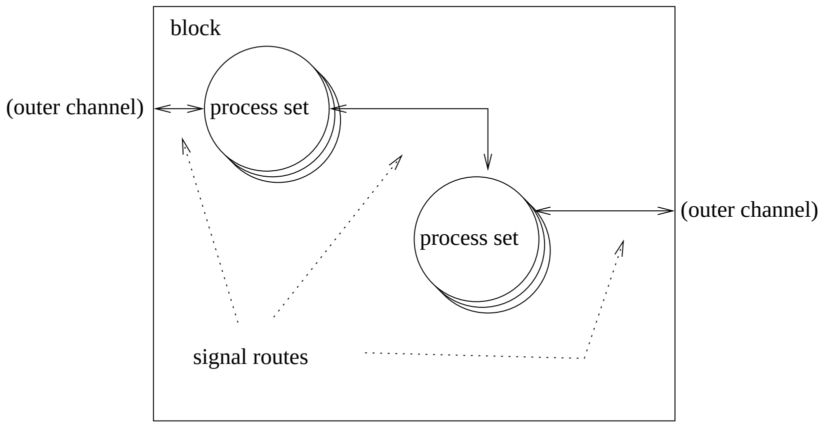


Figure 6: Block with process sets.

**Process level** The dynamic behaviour of an SDL system is due to **process**es. A BNF for **process** in $\mathcal{SDL}_{xta}$ is defined in Section B.9.

## 2.5 Communication

**Signals** A **signal** can be declared in a system, a block, a substructure or a process. A signal must be declared in the highest of the scopes that uses it. A signal can have parameters. A BNF for **signal** in $\mathcal{SDL}_{xta}$ is defined in Section B.3.

**Channels** Channels are used in systems and substructures. A channel has two endpoints, these endpoints can either be connected to blocks or to the **env**ironment. A channel can be uni-directed or bi-directed. The **with** construct restricts, for each direction, the set of signals the channel can convey. The **via** construct refers to a **gate**, which will be discussed in Section 2.6. Channels are delayed non-deterministically. This means that when a signal is sent through a channel there is no guarantee when it is delivered to the receiver, but as the channels are FIFO all sub-sequent signals must also be delayed until the first one is delivered. Since SDL-92, it is possible to use **nodelay** to declare a non delaying channel. In our study all channels are treated as if they were declared as non delaying channels. A BNF for **channel** in $\mathcal{SDL}_{xta}$ is defined in Section B.4.

**Signal routes** Inside a leaf-block the communication channels are called **signal routes**. Signal routes are similar to channels. The **with** and the **via** constructs work in the same way as for channels. Signal routes are never delayed. A BNF for **signal routes** in $\mathcal{SDL}_{xta}$ is defined in Section B.4.

**Connect** The **connect** construct is used to connect an inner and an outer communication channel path in a block or block substructure. In a leaf-block, the inner communication channel will be a signal route. A BNF for **connect** in $\mathcal{SDL}_{xta}$ is defined in Section B.7.

## 2.6 Types, instance sets, gates and references

**Types** There is a distinction between instances (or set of instances) and types in SDL descriptions. It is possible to define **type**s of processes and blocks[2]. Types are not connected (by channels or signal routes) to any instances; instead type definitions introduce gates. These are connection points on the typebased instances for channels and signal routes.

**Instance sets** One can declare an initial and maximal number of instances for a process. This is the reason why we wrote process set instead of process instance in Figure 6. The default is; initial one and maximal one. When more than one process instance is active there is an ambiguity which instance that will receive a signal, if the signal isn't addressed by process id (see Section 2.10). Process instances can be started (**create**d) by other instances and they can **stop** themselves. It is also possible to declare a number of instances for a block. Blocks are static and the number of instances cannot change during run-time.

[2]Not system in $\mathcal{SDL}_{xta}$

**Gates** A **gate** is a connection point used instead of **connect** in typebased instances (blocks or processes). When using types, channels and signal routes adds a "**via** gatename" on an endpoint connecting a boarder of a typebased instance (-set). At a gate one can restrict the set of signals going into or out from the block (or process).

**References** System, block, block type, process and process type specifications can be **referenced** (see Appendix B.5). A referenced specification is a specification placed in the global area that shall be considered to be at the place of reference.

**Scoping rules** A process with the same name can be specified in different leaf-blocks without any naming conflict. When block types, process types and signals are defined they can only be used where they are defined or in a sublevel to that specification. This is like an ordinary programming language with static scoping, e.g., Pascal.

## 2.7 Graphical and texual representation

In the BNF (see Section B) we use only SDL's textual representation (SDL/PR). This is natural because $\mathcal{SDL}_{xta}$ is based on the textual representation and it is $\mathcal{SDL}_{xta}$ that the transformation program in our study uses. SDL also has a graphical representation (SDL/GR) which we not define formally here. We use it to illustrate structure and behaviour.

Figure 7 shows symbols for block, process and their types.

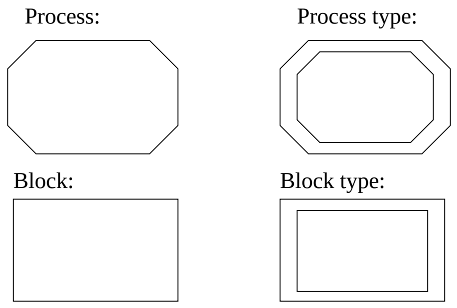


Figure 7: Process, process type, block and block type.

## 2.8 States and input behaviour of a process

SDL's process flowcharts are probably the most commonly known part of SDL. An SDL process can be thought of as a finite-state machine. It has a set of states and for each state it has a set of possible input signals. When one of the possible input signal arrives, the process moves from one state to another (possible the same) state. Such move is called a transition. A transition is a sequence of actions leading to another SDL-state. Transitions are described in Section 2.9. Signals can be saved in a state, that means; if a signal is in the save-set of a state, then the signal can stay in the queue and other (later arrived) signals can be consumed. An arriving signal that is not in the save-set and has no transition specified for it, is discarded.

**States** In a state, the process waits for an incoming signal. The process queries the queue for the next signal and decides which transition it shall follow. A state is declared with the **state** keyword in a textual description. The initial state is an anonymous state declared with the **start** keyword in a textual description. In a state the inputs with their transitions and the save-set are declared. It is possible to specify some inputs for more than one state by emumerating the states that the inputs shall be connected to. The asterisk wildcard ***** can be used following by a list of states inside parentheses indicating that the inputs for this "state" shall be considered as input for all states in the process except those inside the parentheses.

**Input** The graphical symbol of an input is shown in Figure 8. One or more signals can be in the input symbol, indicating that the following transition will be performed for any of the signals. Inputs are declared with the **input** keyword in a textual description. The asterisk wildcard ***** can be used instead of specifying a signal or a list of signals. The asterisk means that for this state all signals except those declared at other inputs shall trig the performance of the transition that follows the input.

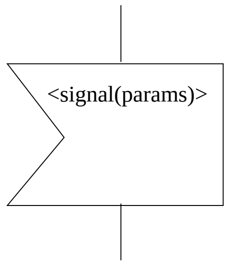


Figure 8: The SDL input symbol.

**Save** In a state a save-set can be declared that is saved as long as the process stays in the state. Save-sets are declared with the **save** keyword in a textual description. The graphical symbol for save is shown in Figure 9.

The asterisk wildcard ***** can be used instead of specifying a signal set to save. The asterisk means that for this state all signals except those declared at other inputs are in the save-set.

**Conditional and continuous input** It is possible to declare an enabling condition on an input. Conditions are declared with the **provided** keyword in a textual description. If no signal is given for an input then the input is a *continuous* input. Such an input must be

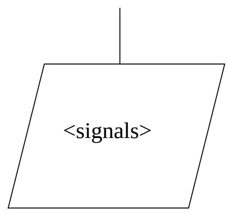


Figure 9: The SDL save symbol.

guarded by a condition. A *continuous* input transition can be performed first when there are no other signals in the queue. The guarded input is not yet implemented in $\mathcal{SDL}_{xta}$.

**Priority** Signals with priority are treated differently in the input-queue. A signal with higher priority is sorted before other signals with lower priority (or no priority) in the input-queue. To set priority the **priority** keyword is used in a textual description. Priority 1 is higher than 2 etc. Priority is not implemented in $\mathcal{SDL}_{xta}$.

**Spontaneous transition** If the signal for an input have is named **none** then the succeeding transition is a spontaneous transition i.e., the succeeding transition can be activated without any stimuli for the process. This will make the state-machine non-deterministic. Spontaneous transitions are not implemented in $\mathcal{SDL}_{xta}$.

## 2.9 Transition

Formally, a transition consists of a sequence of *action statements* with an optional *transition terminator* or just a *transition terminator*. *Action statements* and *transition terminators* will be explained in Section 2.10 and Section 2.11. If no terminator statement is given, then the process stays in the same state. The BNF of the transition, the action statement, and the terminator statement can be found in Appendix B.12.

**Action statements** As action statements we count tasks, outputs, decisions, set and reset of timers, and creation or processes. A task is used to do calculations, an output sends a signal to another process, a decision split the control into multiple paths depending on data, set and reset controls a timer, and create starts a new process instance.

**Terminator statements** As terminiator statements we count nextstate, join, and stop. Nextstate decides the process next state, join connects transfer the control to a particular place in another transition, and stop stops the execution of the current process instance.

## 2.10 Action statements

An action statement is an optional label followed by a task, an output, a decision, a set or reset of a timer, or a creation of a process. We will explan set and reset of timers in

Section 2.12 and creation of processes in Section 2.13.

**Label** A label marks a point in a transition where another transition can join by referring to the label name. A label is an identifier name followed by a colon.

**Task** In a task some computational work can be done, procedure[3] can be called, variables can be assigned etc. External actions can also be made here often expressed as informal text. Figure 10 shows the graphical symbol of a task.

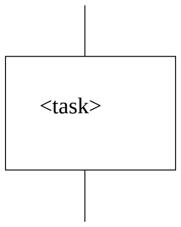


Figure 10: The SDL Task symbol.

**Output** Communication between processes is done by sending messages. When a process sends a message it uses the action output. The keyword in the textual representation is **output**. In the textual representation the output construct consists of the **output** keyword, a signal identifier with optional parameters, an optional **to** construct, and an optional **via** construct.

The to construct consists of the keyword **to** followed by an identifier of the process to send to. The identifier could be a variable of type PId or a name of a process-set. The former includes the special SDL PId variables **self**, **sender**, **parent**, and **offspring**. Where self is the process' own pid, sender is the pid of the last handled signal, parent is the pid of the creator of the process, and offspring is the pid of the last created child created by the process. All except self variable can have value zero. The sender variable if the process hasn't received any signal, the parent if the process was created at initialization, and the offspring if no child has been created. The via construct consists of the keyword **via** followed by one or more identifiers. An identifier can be a gate, a signal route, or a channel. The purpose of the via is to show the path out from the process and thereby constrain the receiving process-sets. Figure 11 shows the graphical symbol of an output.

**Decision** A transition may split into two or more branches depending on some condition using the **decision** construct. The decision construct consists of a question and a decision body inside a **decision enddecision** pair. The decision body consists of answers, each of them has a continuing transition. One of the answers can be an **else** answer which is then the supplement of all other answers. After the **enddecision** all transitions that not

[3]Procedures are not implemented in $\mathcal{SDL}_{xta}$.

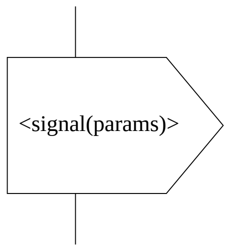


Figure 11: The SDL output symbol.

end with a terminator statement are implicitly joined and continues with the consecutive transition. If the question of the transition is a any question, then all answers are empty and one of the answers is chosen randomly at execution. Figure 12 shows the graphical symbol of a decision.

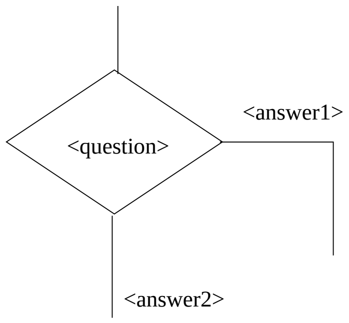


Figure 12: The SDL decision symbol.

## 2.11 Terminator statements

Nextstate, join, and stop are terminator statements. The stop statement will be described in Section 2.13.

**Nextstate** When a transition has been performed, then the control returns to a "stable" state, where the process waits for a new input signal. A nextstate construct consists of the **nextstate** keyword and an identifier for the next state. If an hyphen "-" is given instead of an identifier, control stays in the same state for the process.

**Join** Join has the same effect as a goto in a programming language. It directs the flow of control to a point that is marked by a label. This name is given in the join construct after the **join** keyword.

## 2.12 Timers

A timer is declared (defined) inside a process. At run-time each process instance has its own set of the declared timers. A timer is set to expire at a certain time in the future. When a timer expires, it send a signal (named after the timer) to its process' queue. It becomes active when it is set and becomes inactive when it is reset or when its signal is consumed by the process. The timer's signal is put on its process' queue like all other signals, so it is possible for the process to set or reset the timer after a timer signal has been put on the queue, but before it has been consumed. In this case the timer signal is removed from the queue. This makes it impossible for the process to get the timer signal twice from the single set occasion or to get a timer signal after it has reset the timer.

**Set** The set statement consist of the **set** keyword and two arguments inside parentheses. In the first argument is a time expression and the second the name of the timer. The time expression is calculated in run-time to the time when the timer shall expire and send its timer signal. It usually contains an offset to **now**. The first argument can be omitted. In that case the timer expires at time "**now** + duration" where duration is derived from the timer definition.

**Reset** The reset statement consist of the **reset** keyword and inside parentheses the timer to be reset. After a timer is reset it cannot expire until it has been set again.

## 2.13 Process management

**Create** A create is used to spawn a new process instance of a process set. The create construct consists of the **create** keyword followed by the name of the process set wherein the new instance shall be made. After a process instance has been created the parent process gets the process id of its child process in a predefined variable named **offspring**. At startup, the child process gets its creators process id in the variable **parent**.

**Stop** When a process instace has done its work it can terminate itself with the stop statement. The **stop** keyword is used for this purpose. Figure 13 shows the graphical symbol of a stop statement.

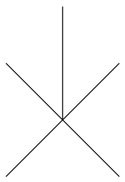


Figure 13: The SDL stop symbol.

# 3 Uppaal

## 3.1 History

Uppaal is an integrated tool environment for modeling, simulation and verification of real-time systems, developed jointly by Basic Research in Computer Science at Aalborg University in Denmark and the Department of Computer Systems (DoCS) at Uppsala University in Sweden. Uppaal's first release was in 1995, and it has been applied in a number of case studies.

## 3.2 Definition

Uppaal consists of three main parts: a description language, a simulator and a model-checker. The description language is a non-deterministic guarded command language with simple data types (e.g. bounded integers, arrays, etc.). It serves as a modeling or design language to describe system behavior as networks of automata extended with clock and data variables. The simulator is a validation tool which enables examination of possible dynamic executions of a system during early design (or modeling) stages and thus provides an inexpensive mean of fault detection prior to verification by the model-checker which covers the exhaustive dynamic behavior of the system. The model-checker can check invariant and reachability properties by exploring the state-space of a system, i.e., reachability analysis in terms of symbolic states represented by constraints.

## 3.3 Modeling in Uppaal

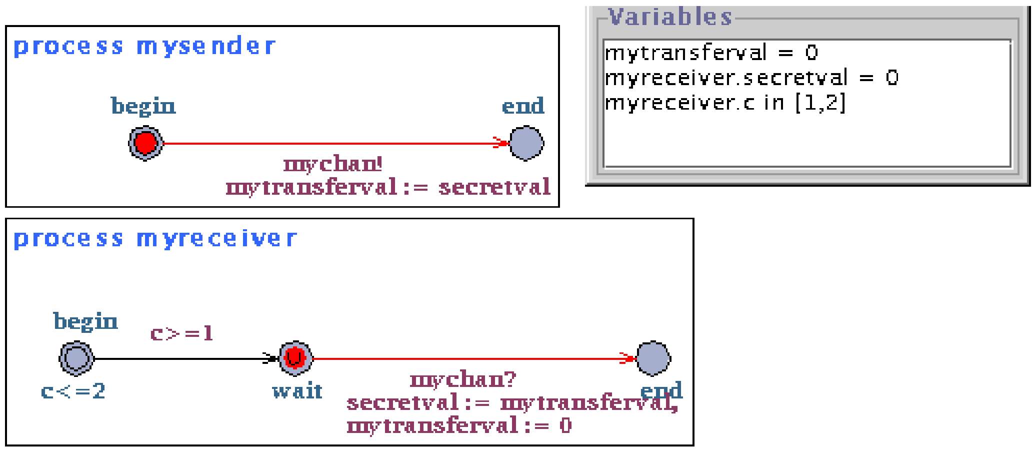
When we take a closer look at the modeling language it is important to remember that the model is not executed but rather searched by the simulator or the model-checker. In the simulator we choose one of the possible transitions. All clocks in Uppaal have fictive simulated time which has nothing to do with the wall clock or any other time counting unit. We will come back to this discussion later.

```
/////////////////////////
// Global declarations
int mytransferval := 0;
urgent chan mychan;

/////////////////////////
// Templates
process Receiver(urgent chan schan; int stransfer)
{
 // local declarations
 int secretval;
 clock c;

 // definition of locations
 state begin, wait{c<=2}, end;
 // definition of urgent locations
 urgent begin;
 // The initial location
 init begin;
 // Transitions
 trans begin -> wait {
   assign c := 0;
 },
 wait -> end {
  guard c>=1;
  sync schan?;
  assign secretval := stransfer, stransfer := 0;
 };
}

process Sender(urgent chan schan; int stransfer)
{
 const secretval 3;
 state begin, end;
 urgent begin;
 init begin;
 trans begin -> end {
  sync schan!;
  assign stransfer := secretval;
 };
}

/////////////////////////
// Process assignments (instantiation of templates)
mysender := Sender(mychan, mytransferval);
myreceiver := Receiver(mychan, mytransferval);

/////////////////////////
// System definition
system mysender, myreceiver;
```

Figure 14: Textual description of sender receiver.

## 3.4 Basics

An UPPAAL system is a collection of automata communicating with channels for synchronization and variables for sharing values. Each automaton has control nodes and transition edges describing its behaviour. The system's current state is defined by the processes locations, the variable and the clock values.



```
mytransferval = 0
myreceiver.secretval = 0
myreceiver.c in [1,2]
```

Figure 15: Simulation of UPPAAL text in Figure 14.

**System, Templates and Processes.** In UPPAAL's .ta format a system is declared with the keyword **system** follwed by the names of the processes included in the system. Global variables used in processes must be referred to directly. There is a one-one correspondence between a declared process and an instance in the system.

In the newer .xta format not processes but templates are defined e.g., *Receiver* in Figure 14, still the keyword **process** is used. A template is a process type with zero or more parameters. A process (as in the .ta format) is instantiated by parameterizing a templete. In simulation/verification time the .xta format will have been translated into the older .ta format. The result is shown in Figure 15. Therefore any reference to Figure 14 is also a reference to Figure 15.

**Locations.** Locations are the control nodes of an UPPAAL process. In Figure 14, the *Receiver* template has control nodes $\{begin, wait, end\}$. After the **state** keyword all the process states are listed. The *begin* location is the *Receiver*'s initial location defined with the keyword **init**.

**Transitions.** A transition is an edge between two control nodes (locations) in an UPPAAL process. A transition may be guarded by a timing or data constraint, it may require a synchronization with another process and it may include a clock or data assignment. After the **trans** keyword the transitions are listed for the process. A transition goes from one location to another e.g., *wait* to *end* in Figure 14.

**Guards.** Guards express conditions on the values of clocks and integer variables that must be satisfied for the transition to be taken. The keyword for guards in a transition is **guard** e.g., **guard** $c >= 2$; in Figure 14. See Section C for syntax on integer guards.

**Synchronization.** Synchronization means that two UPPAAL processes change location in one simulation step. Synchronization is done with (or via) channels. Channels are declared with the **chan** keyword e.g. *mychan* in Figure 14. To synchronize two processes a channel is specified after the **sync** keyword followed by a "**!**" for one of them and a "**?**" for the other, e.g. process mysender at location *begin* has a transition to location *end* with **sync** mychan**!** and process myreceiver at location *wait* has a transition to location *end* with **sync** mychan**?**. Figure 14 shows templates that will be translated to this.

**Assignments.** Assignments can be done to integer and clock variables in transitions. An assignment is declared with the keyword **assign**. Clock variables can, in the current implementation, only be assigned with constant values. For an exact definition of assignments on integer variables see Section C. After the **assign** keyword a comma separated list of assignments is given, mixing assignments for integer and clock variables is no problem. In Figure 14 the mysender will communicate its *secretval* to myreceiver by assigning the global variable *mytransferval*, locally called *stransfer* in both mysender and myreceiver.

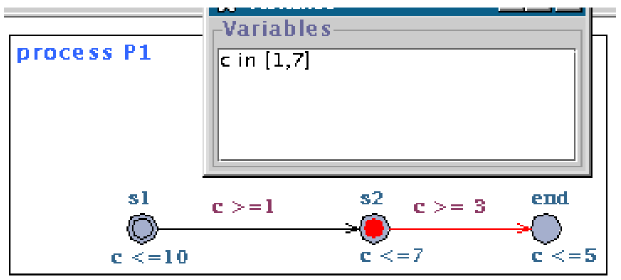
UPPAAL processes can communicate in two ways: by sharing data and by synchronization. Globally declared data is readable and writable for all processes, e.g. *mytransferval* in Figure 14.

## 3.5 Enforcing progress in the system

Without considering timing an automaton has the possibility to wait forever in a control location. Happily timed automata provide a number of ways to prevent this from happening.

- Invariants
- Urgent Locations
- Committed Locations
- Urgent Channels

**Invariants.** An invariant is a constraint on the clock values in a location. This put constraints on the simulator and the model checker. Consider Figure 14, where the *Receiver* has an invariant $\{c <= 2\}$ on its *wait* location. The meaning of this is: The automaton can remain in location *wait* only as long as $\{c <= 2\}$. The transition from the *wait* location to the *end* location can only be taken when $1 <= c <= 2$, expressed as "$c$ in $[1, 2]$". This is shown in Figure 15.



c in [1,7]

Figure 16: Possible, but not always.

In Figure 16 we can study how timing constraints work. The control is at location $s2$. When leaving location $s1$ we know by now that we have "$c$ in $[1, 10]$". When we enter location $s2$ we have "$c$ in $[1, 7]$" this is because of the invariant $\{c <= 7\}$ on the location $s2$. Are we turning time backwards? No, the simulator observes that this transition is valid only when we have "$c$ in $[1, 7]$". Cases where "$c$ in $]7, 10]$" (that is $7 < c <= 10$), are no longer valid after the step. But what if we have "$c$ in $]7, 10]$"? Don't we want to know about this potential model error? If we try to prove, with the UPPAAL's verifier, that we always can go from $s1$ to $s2$, then we will get a negative answer.

In Figure 17 we must have "$c$ in $[9, 7]$" because of the synchronization between the two processes $P1$ and $P2$. Though this is an impossible step the simulator will tell us that we have reached a deadlock.

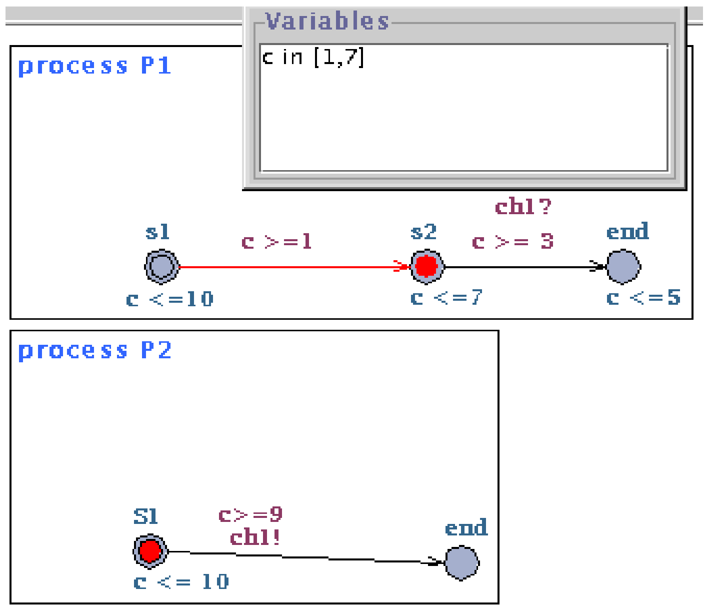


Figure 17: Impossible synchronization. $c$ cannot be in $[9, 7]$.

**Urgent Locations.** "Urgent locations" are locations that the control must leave without any delay, as if we had an invariant $\{c <= 0\}$ on the location and "**assign** c:=0;" on all incoming edges. This forces the actual process to always make a transition without any delay. Urgent locations are listed after the keyword **urgent** e.g, the *wait* state in Figure 14.

**Committed Locations.** "Committed locations" are even more restricted than Urgent Locations. No other process is allowed to make any simulation steps without being synchronized with the committed one. Committed locations are listed after the keyword **commit**.

**Urgent Cannels.** When a channel is declared as an urgent channel, then synchronizations via that channel has priority over normal channels and the transition must be taken without delay.

# 4 SDL run-time system building

When writing an SDL description we model a specification where processes are described in terms of process types, process definitions in a hierarchy of a system and blocks. In order to simulate the description, we must then convert the SDL description to an **instance domain**, we must also design a run-time system which manages process instances and communication. The interpretation of structure and communication from the SDL description is handled by SDL's meta processes such as *system*, *timer* and *input-queue*. The discussion of converting the behaviour of an individual process such as input, output, decisions and queuing, will be deferred to Section 5.

In this section we will outline how UPPAAL automata can work in a structure to simulate an SDL run-time system. We will also look at our implementation of SDL's meta processes in UPPAAL. This includes our solutions for: timers, dynamic process creation, and signal delivery.

## 4.1 $\mathcal{SDL}_{xta}$ run-time overview

Each SDL process instance will have its own queue instance and a timer instance for each declared timer. Consider Figure 18, we have a dashed line, labeled *Process Instance* (*PI*) around the UPPAAL processes that are generated for each SDL process instance. We parameterizing UPPAAL templates in a appropriate way.

If we declare more than one instance of an SDL process, we get a process set. A process set is a collection of process instances, which differs only by their process id (pid). Signal routes that are connected to a process set are equally connected to all process instances. If an incoming signal is not addressed to a specific pid then each created process instance has equal opportunity to receive the signal. We simulate this signal ambiguity by means of an UPPAAL process called *process set input* (*PSI*), common for all instances in the process set.

When any process instance, in a process set, is sending a signal without specifying the receiving pid, then there can be more than one possible receiving $PSI$. This ambiguity is due to underspecification. To simulate this ambiguity we create, for each process set, an UPPAAL process called *process set output* $PSO$. In an SDL output it is possible to restrict the receiving $PSI$s by using SDL output's via or to construct. The task of the $PSO$ is that for each output, it should route the signal to the right $PSI$. If there is more than one possible $PSI$ then it shall chose one randomly. The $PSO$ is shared by all process instances in a process set, just like the $PSI$. Figure 20 shows the UPPAAL processes that are involved at signal delivery.

When a signal has been sent and the receiving pid is specified the signal goes via a global UPPAAL process called *Expl*. When *Expl* receives a signal it send the signal to the queue with the specified pid. We will discuss this more thoroughly in Section 4.2.

In UPPAAL there are a fixed number of automata during the simulation/model check. To simulate dynamic creation and termination of processes we add locations where the automata are considered inactive. When an SDL process is about to create a new instance of another SDL process, it communicate with the $PSI$ of the process set where the new

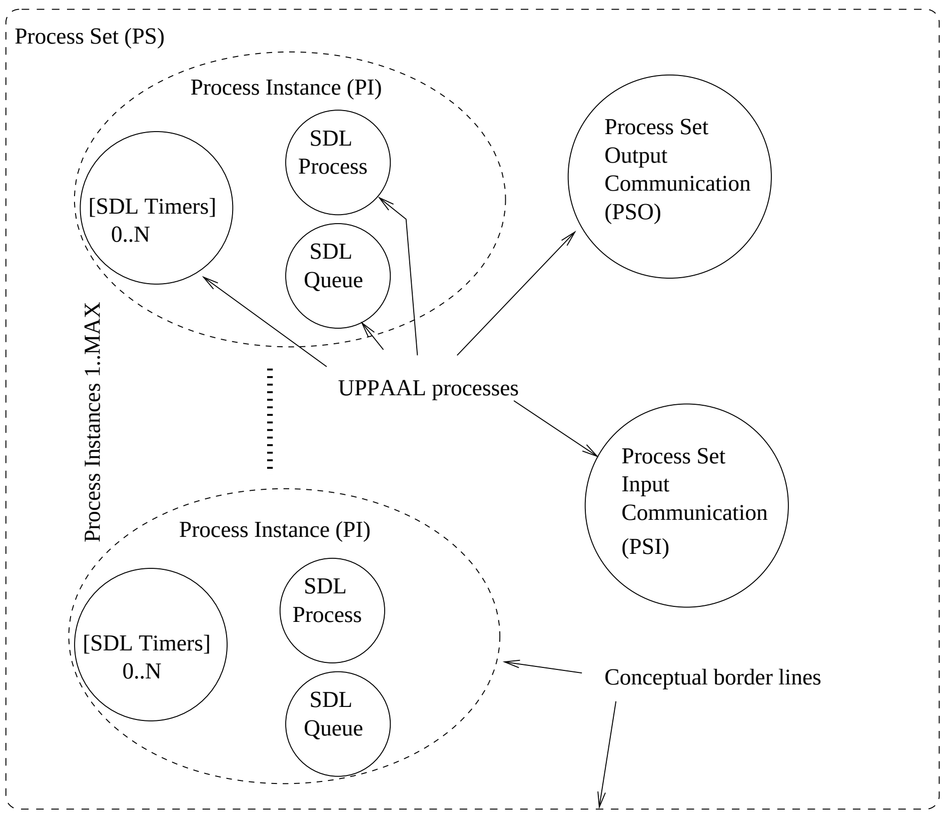


Figure 18: Generate of a process set.

instance is to be created. The $PSI$ will, if there is any inactive instance left, activate one. In SDL we would get a new pid for each created process instance. This would make the search space for Uppaal infinite and as a consequence make it impossible for Uppaal's model checker to prove anything for all system states. Our solution is to reuse pids. An instance keep its pid even when it is inactive. By removing all knowledge of a pid in the system when the process instance terminating, we achieve an almost correct pid semantics. The work of removing a system's knowledge of a pid is done by an Uppaal process called *RemPid*. Figure 24 shows the Uppaal processes that are involved in process instance creation.

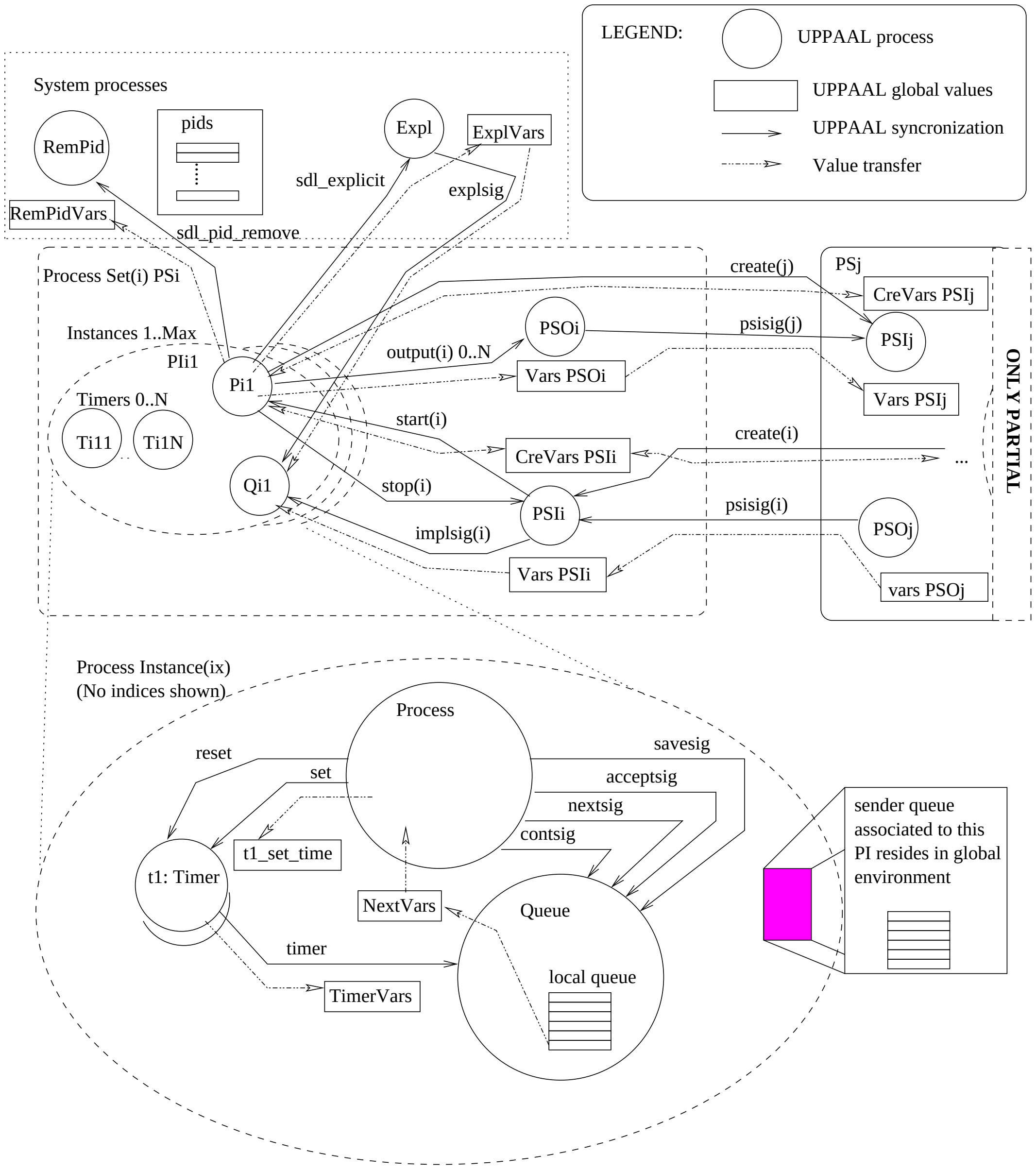


Figure 19: Ta run-time system.

In Figure 19 we show all types of processes, Uppaal synchronizations, and data storage in our $\mathcal{SDL}_{xta}$run-time system. We henceforeward simplify the notation of the Uppaal synchronization channels and variables by only using their functional names and thereby omitting the scoping prefix. The valid scope for each name is shown in Table 1 and Table 2. We give three examples to shed some light on this.

Firstly, when a process synchronizes over a `nextsig` channel only the queue in the same process instance can synchronize with it, i.e., a `nextsig` channel has a prefix so it is only "shared" inside a process instance.

Secondly, when a *PSI* synchronizes over a `implsig` channel any *PI*'s queue process in the process set can synchronize with it, i.e., a `implsig` channel has a prefix so it is only shared inside its process set.

Thirdly, when the *Expl* (there is only one) synchronizes over the `explsig` channel any queue can synchronize with it, i.e., the `explsig` channel is globally known.

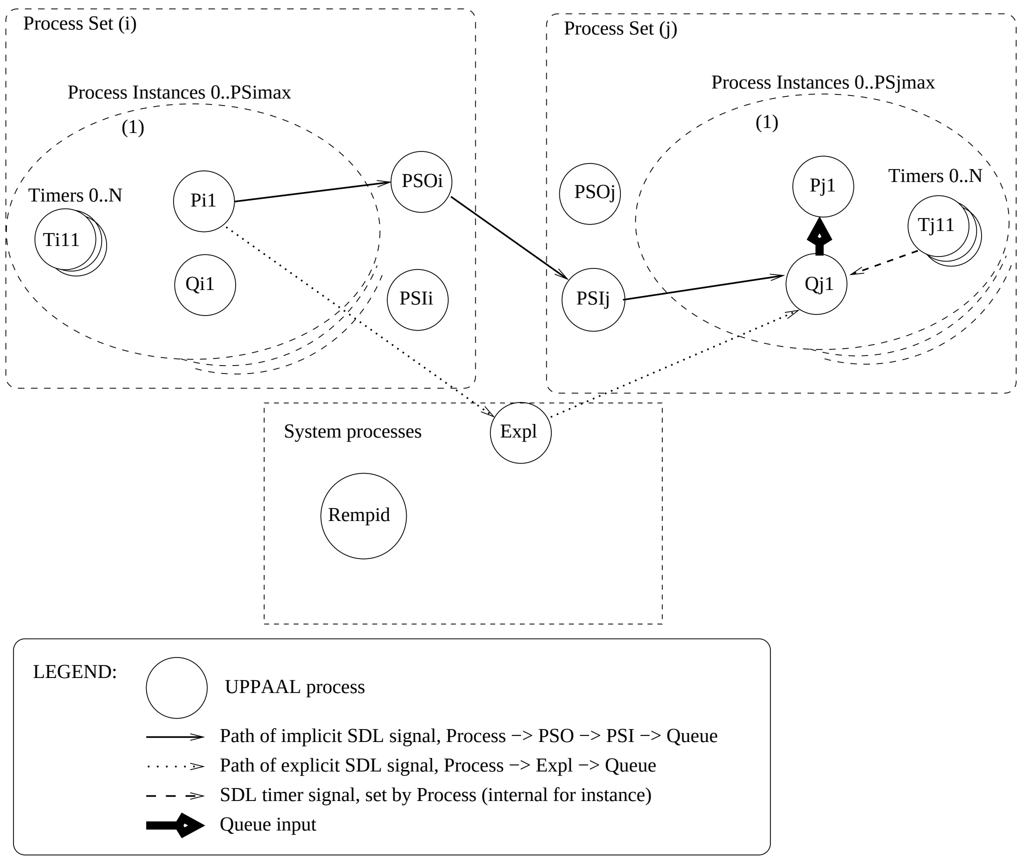
| | | |
|---|---|---|
| **Data storage globally shared** | | |
| RemPidVars | `rem_pid` | Pid for removal |
| Other global | `qlive[]`<br>`pids[]` | Bitarray for activation control<br>Storage for pids |
| ExplVars | `expl_signal`<br>`expl_params[]`<br>`expl_sender`<br>`expl_to` | Signal<br>Parameters (array)<br>sender's pid. Index to `pids[]`<br>Destination pid, explicit addressing |
| **Data storage shared for a Process Set (PS)** | | |
| Vars PSO | `pso_signal`<br>`pso_params[]`<br>`pso_sender` | Signal<br>Parameters (array)<br>Sender's pid. Index to `pids[]` |
| CreVars PSI | `ps_offspring`<br>`ps_parent` | Index in `pids[]` to parents **offspring**<br>Index in `pids[]` to parents pid |
| Vars PSI | `psi_signal`<br>`psi_params[]`<br>`psi_sender` | Signal<br>Parameters (array)<br>Sender's pid. Index to `pids[]` |
| **Data storage shared for a Process Instance (PI)** | | |
| NextVars | `in_signal`<br>`in_param[]` | Signal<br>Parameters (array) |
| Other PI Vars | `self`<br>`parent`<br>`offspring`<br>`sender` | PI's pid, constant<br>PI's creator's pid. Index to `pids[]`<br>Last child's pid. Index to `pids[]`<br>Sender of last signal's pid. Index to `pids[]` |
| For each timer:$t$ | `t_set_time` | Value timer $t$ is set to |

Table 1: Explanation of storage from Figure 19.

| UPPAAL **channels globally shared** | |
|---|---|
| `sdl_pid_remove` | Syncs PI and *RemPid* when PI stops |
| `sdl_explicit` | Syncs a sending PI and *Expl* when explicit addressing |
| `explsig` | Syncs *Expl* with the queue of the PI with the right pid at explicit addressing |
| UPPAAL **channels shared for a Process Set (PS)** | |
| `output` | Syncs PI and *PSO*, one "output"-signal for every different SDL output implicit addressing |
| `psisig` | Syncs external PS's *PSO* with channel's *PSI*. Deliver signal |
| `implsig` | Syncs *PSI* with PI's queue |
| `create` | Syncs external PS's *PSO* PI with signal's *PSI* |
| `start` | Syncs *PSI* with PI, to start the PI |
| `stop` | Syncs PI with *PSI*, when PI stops |
| UPPAAL **channels shared for a Process Instance (PI)** | |
| `savesig` | Process decide to save signal |
| `acceptsig` | Process decide to save signal (or discard) |
| `nextsig` | Process gets next signal from queue |
| `contsig` | Process gets OK from queue to take the continous signal |
| `timer` | Syncs timer with queue |
| `set` | Process sets timer (one per timer) |
| `reset` | Process resets timer (one per timer) |

Table 2: Principal UPPAAL channels in run-time system.

## 4.2 Signal delivery



Process Set (i)
Process Instances 0..PSimax
(1)
Timers 0..N
Ti11
Pi1
Qi1
PSOi
PSIi
Process Set (j)
Process Instances 0..PSjmax
(1)
PSOj
PSIj
Pj1
Qj1
Timers 0..N
Tj11
System processes
Expl
Rempid
LEGEND:
UPPAAL process
Path of implicit SDL signal, Process –> PSO –> PSI –> Queue
Path of explicit SDL signal, Process –> Expl –> Queue
SDL timer signal, set by Process (internal for instance)
Queue input

Figure 20: Ta signal paths.

**Explicit signal delivery** For explicit delivery of signals the *Expl* UPPAAL process is used. There can be only one *Expl* process in the whole system. The *Expl* UPPAAL process is shown in figure 21. If an output action of a process explicitly sends a signal **to** a known pid, then there is a unique receiver. The sending process instance, synchronizes with the *Expl* process over UPPAAL channel `sdl_explicit` which is globally known. In that synchronization, the process transfers data to unique global locations used by *Expl*, see *ExplVars* in Table 1. After this synchronization, *Expl* delivers the signal to the process instance with the destination pid. This synchronization uses the `explsig` UPPAAL channel. It is the input-queue of the process instance that receives and stores the data. There is a boolean value in a global array called `qlive[]` where the instances track if they are active or not. Instead of waiting for a process instance queue to synchronize, the *Expl* UPPAAL process discards the message if the receiving queue is inactive.

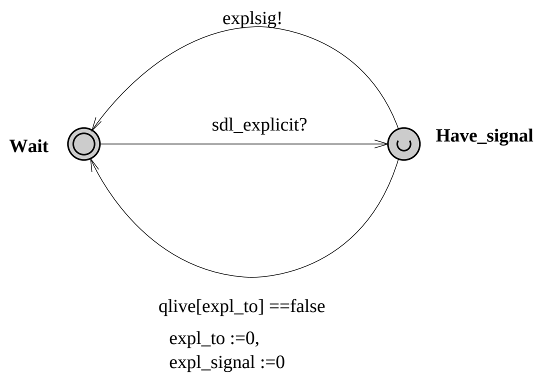


Figure 21: The *Expl* ta process.

**Implicit signal delivery** When the pid for the receiving process instance is not known i.e., it is not given by the output action, the signal will take a different path to its destination queue. We will refer to such output as **implicit** signaling. Implicit outputs can be ambiguous. We have to simulate this ambiguity in the generated code. The signal analysis, described in Section 4.4 will for every output decide the possible process sets where the signal can be received. In a run-time system, this knowledge resides in the *process set output* ($PSO$) of the sending process instance.

An implicit output of an SDL-process is handled as follows. Firstly, the sending SDL-process synchronizes with its $PSO$ process over an UPPAAL channel named after the signal and the contraints of the output. The $PSO$ serves all process instances which shares this UPPAAL channel. In that synchronization the sending process instance transfers data to its $PSO$'s storage called *Vars PSO*, described in Table 1. After this synchronization, the $PSO$ synchronizes with one of the $PSI$s that is in the possible destination set. Here the sending $PSO$ uses the receiving $PSI$'s data storage, *Vars PSI*, the $PSO$ and the $PSO$ synchronize over $PSI$'s UPPAAL channel for incoming signals. The $PSI$ is unaware of which $PSO$ it is communicating with. Lastly, the $PSI$ transfers the signal to one of the process instances' queue. The $PSI$ is serving the queues for all process instances in the process set, and when a signal is arriving via an implicit path, the receiving process instance is any of the (running) process instances. To get the desired effect we use ambiguous UPPAAL synchronization. That is, the $PSI$ synchronizes with one of the queues. The queue reads the values from the $VarsPSI$.

To explain the functioning of the $PSO$ consider Figure 22. We can see that its SDL-process has only one output, namely "**output** `Money` **via** `sr1`"[4]. There are two possible receivers of the signal, `P1` and `P2`. The process set `P1` has two instances. The latter observation comes from the edge that communicates with neither of the process sets. It is guarded with a test where all instances of all possible receiving process sets must be inactive for the transition to be taken. `P1`'s instances are named `P1_1_0` and `P1_1_1`. First we have the name of the process set (P1) then we add a unique number for this process set, in this case 1 and the last figure is an instance counter. For the process set P2 that has only one instance, that instance has the name of the process set.

For the $PSI$ consider Figure 23, it is a generic process. This UPPAAL template is used for all process sets. It is mediating the signals from a sending $PSO$ to one of the input-queues in the $PSI$'s process set. If there is no process instances started, it will not accept any signals. This forces the sending $PSO$ to either send the signal to another $PSI$ or discard it. The create and stop edges are part of the dynamical process management, described in the next section.

[4]It can have more, if they are all exactly the same.

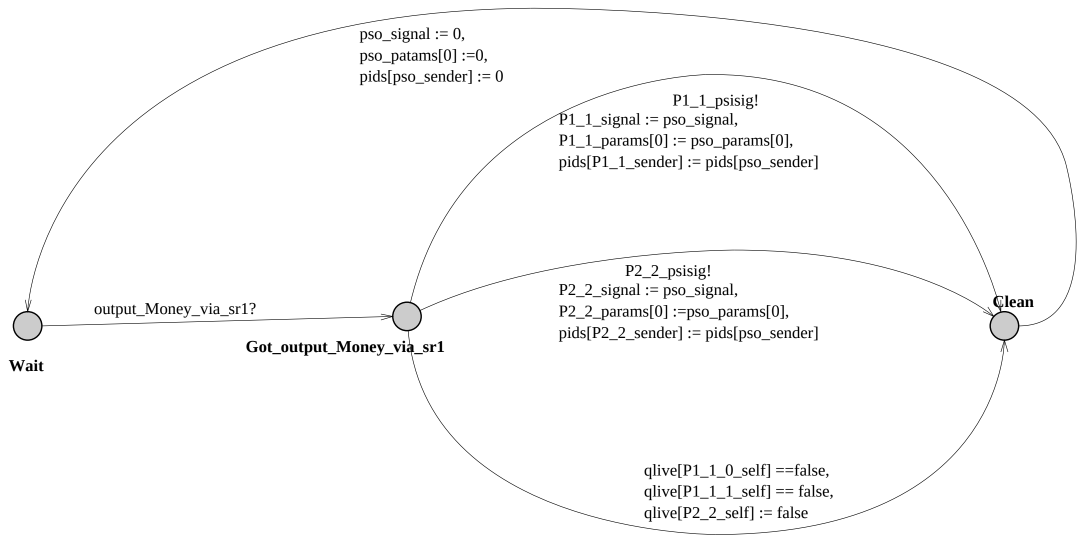

pso_signal := 0,
pso_patams[0] :=0,
pids[pso_sender] := 0
P1_1_psisig!
P1_1_signal := pso_signal,
P1_1_params[0] := pso_params[0],
pids[P1_1_sender] := pids[pso_sender]
P2_2_psisig!
P2_2_signal := pso_signal,
P2_2_params[0] :=pso_params[0],
pids[P2_2_sender] := pids[pso_sender]
output_Money_via_sr1?
Clean
Wait
Got_output_Money_via_sr1
qlive[P1_1_0_self] ==false,
qlive[P1_1_1_self] == false,
qlive[P2_2_self] := false

Figure 22: An example *PSO* Uppaal process.

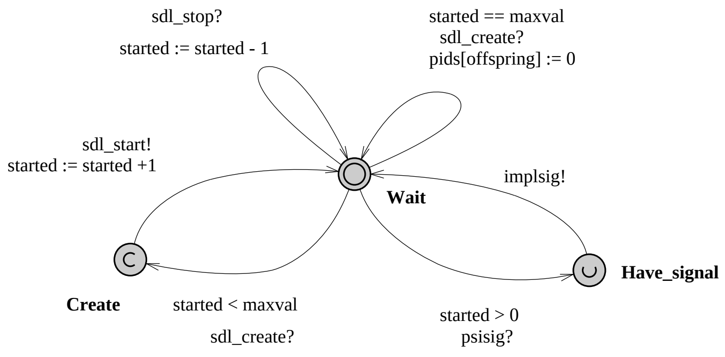


Figure 23: The *PSI* Uppaal process.

## 4.3 Dynamic creation of processes

In this section we describe how we simulate dynamic creation of processes. In UPPAAL we can only use a fixed number of processes. This restricts us to a maximum of processes given at specification. A maximum number of instances can be specified for each process set, stating how many simultaneous instances a process set can have. When a process instance terminates, it gives room for a new instance, etc. Our solution is to generate the maximum number of instances and let them keep thier process id. SDL specifies that a newly created process instance shall have a new and unique process id. To follow this semantics and still keep the same process id we instead let the rest of the system forget about a process id when a process terminates. To get the right number of initial process instances per process set we create an UPPAAL process that starts the appropriate number of instances for each process set. Some instances are active and other are passive. We ensures that signals can be delivered only to an active instance's queue. We also have to make a control signal path from a creating process to the process set where an instance shall be created.

**Process creation.** When an SDL-process in a process set wants to start a process instance in another process set (or even its own), it synchronizes with the *PSI* of the process set it wants to start. Consider Figure 23. When the *PSI* gets a request for a creation of a new process instance and its maximum number of instances is not reached, it does a transition to the **Create** location. In this transition the creator has set the receiving process set's `ps_offspring` value to a reference to its offspring value. Also it has set its own pid in the receiving process set's `ps_parent`. Futher in the transition back from the **Create** location it commence execution of one of the process set's inactive instances. In that transition, the started SDL-process sets its pid to the creator's `ps_offspring` and moves the `ps_parent` to its own **offspring**.

**Start the initial instances.** The StartUp UPPAAL process has a very simple task. It creates the initial number of process instances for each process set. This is done in the same way as when a new process instance is created. For each process set where the number of instances at system start (the initial value) is greater than zero the StartUp UPPAAL process synchronizes with the *PSI* of the process set over the process set's `create` UPPAAL channel, once for each initial instance. Consider Figure 25; it creates on instance for the `p1` process set and one for the `p2` process set.

**Start and stop of an SDL-process.** In Figure 26 we show an excerpt of an UPPAAL SDL-process. At the location **label1** the SDL-process starts and when it stops it goes to location **iSDL_stop**. Note, we are using the **iSDL_** prefix for internal SDL locations i.e., SDL process management. The locations shown are the start up sequence and the termination sequence. The initial state of the SDL-process is called **iSDL_Inactive**. This location simulates that the process instance does not exist. When the *PSI* of the process instance gets a create message, the *PSI* synchronizes with a process instance over channel `sdl_start`. The control moves to **iSDL_Start**. In that move the creating and the created process instances exchanges pids. The *ps_offspring* has been set to the index in `pids[]`

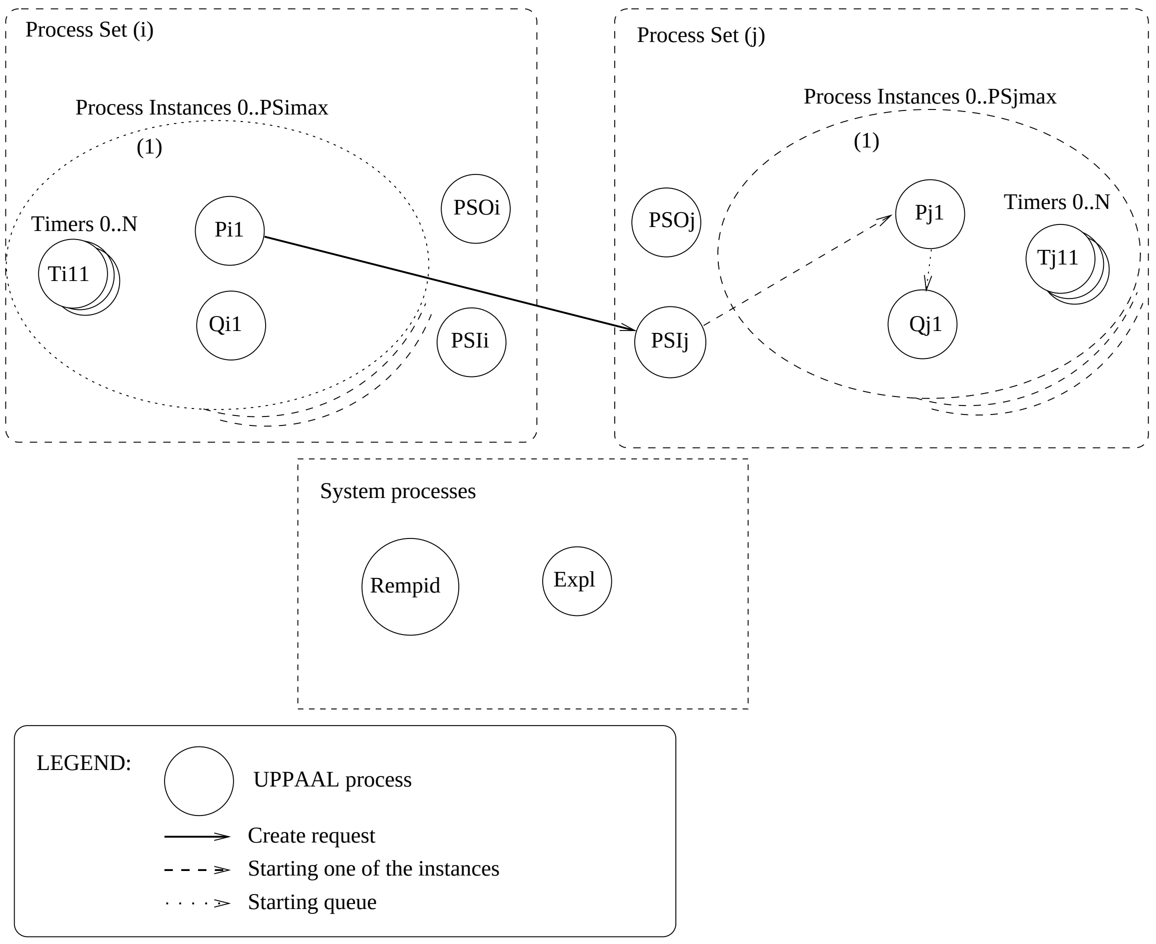


Figure 24: Principles of create.

where the creating process instance has its offspring and the *ps_parent* index in `pids[]` has been set to the creating process instances' pid. In the move from **iSDL_Inactive** to **iSDL_Start** the created process sets the creators offspring (indirected) to its pid (self) and sets its parent (indirected) to the creators pid.

When moving from **iSDL_Start** to the initiation actions the process instance's queue is started by a synchronization, locally called `q_start`.

When an SDL-process stops it goes to the **iSDL_Stop** location, starting a termination phase. When it moves from **iSDL_Stop** to **iSDL_StopQueue** it synchronize with its $PSI$. The $PSI$ counts the number of active process instances and it can now decrease the number by one. When it moves from **iSDL_StopQueue** to **iSDL_RemPid** it synchronizes with its queue. After this synchronization the queue becomes inactive and can't receive any signals. When the control moves from **iSDL_RemPid** to **iSDL_Reset** it synchronizes with the *RemPid* process. In this move it also assigns the global `rem_pid` variable, which

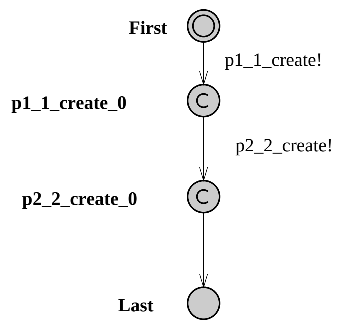


Figure 25: An example *StartUp* UPPAAL process.

decides on what pid the *RemPid* shall work. Last, when moving from **iSDL_Reset** to **iSDL_Inactive** we reset variables to the values that UPPAAL would have given them at system start up.

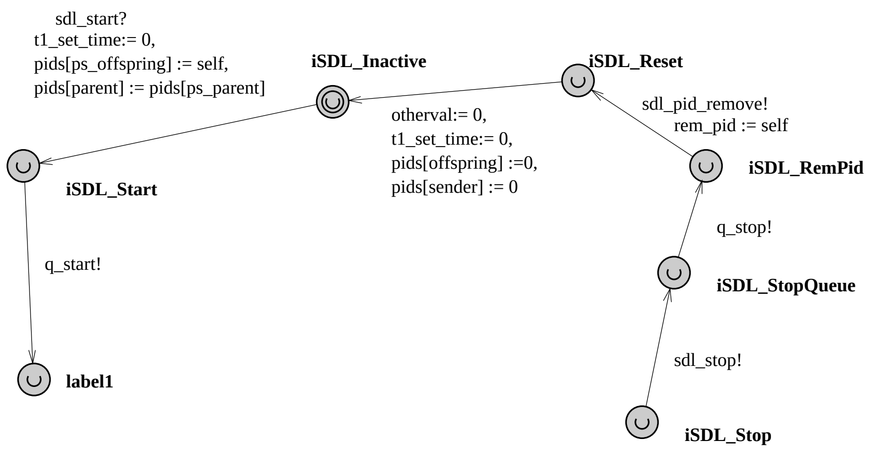


Figure 26: Start and stop principle. Excerpt from an sdlprocess.

**Pid removal.** The *RemPid* UPPAAL process is a global process that is responsible for removing all references to a given pid. Every pid variable (local and global) that holds the pid value of a stopped instance has to be replaced with a zero. By doing this the pid is forgotten by the system outside the instance and can be reused if that instance is restarted. To make this work feasible for *RemPid* all pid values are stored in an array called `pids[]`. In the places where pids are used, instead of holding a pid we hold an index to the pid array where the pid value is stored. There are two exceptions to this rule, sender queues and the `expl_to`. A sender queue is a queue of sender pids corresponding to a queue of signals that a *Queue* UPPAAL process holds. When a signal is sent that signal is associated with the pid of the sender. For every *PI* in the system there will be a sender queue that the *RemPId* has to go through.

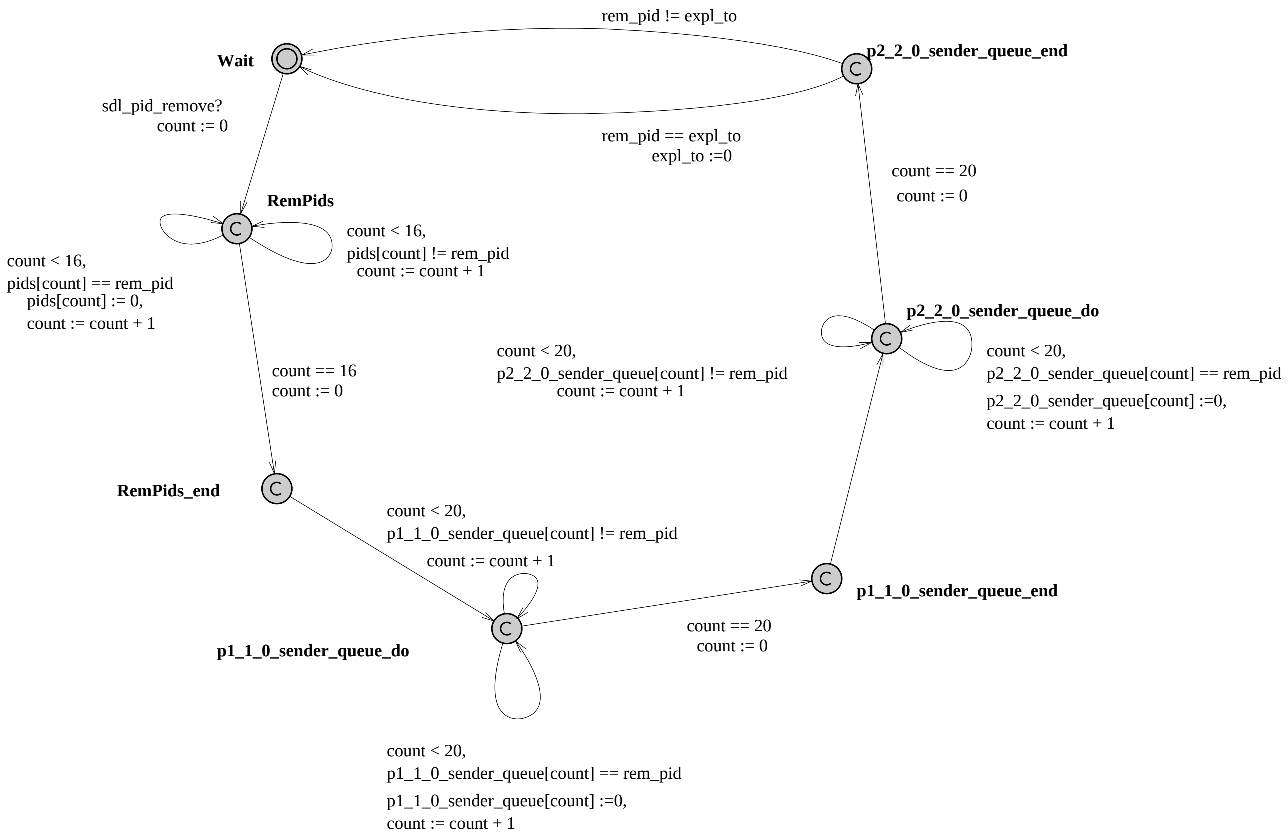


Figure 27: An example *RemPid* UPPAAL process.

An example of the *RemPid* UPPAAL process is shown in figure 27. The *RemPid* starts in the **Wait** location. When an SDL-process synchronizes over the `sdl_pid_remove` channel it also gets the pid to remove in the `rem_pid` data storage. In the **RemPids** location it exchanges all values equal to `rem_pid` to zero in the `pids[]` array. When we have reached the end of the array, in this case `pids[16]`, we break and transit the control to a location called **RemPids_end**. Futher, we do the same cleaning for the sender queues of the processes `p1` and `p2`. Lastly, we clean the `expl_to` value if necessary. All these stages are done using committed locations as to have no interruption while doing this cleanup.

## 4.4 Signal analysis

In this section we present the analysis of *implicit* signal paths. This analysis gives the information for the $PSO$ to know the set of possible destination $PSI$s for each output action of an SDL-process. There are three parameters for each output: signal, via and to, where via and to are optional. If these parameters are exactly the same for two outputs, then they share the same set of possible destination $PSI$s. Our task is to follow signal routes and channels, with the constraints given by the parameters (signal, via, to) to get a set of $PSI$s.

The algorithm is basically to follow every signal route from the process set to the process set it is **connect**ed to, or to the **env**ironment, where we follow the channel(s)[5] that the signal route is connected to, or in the case where the block is typebased and the signal route is connected to a gate, we follow the channel(s) connected to that gate. When following a channel into a block we can also have more than one continuation. Absence of specification is treated liberally. This means that if no signal routes/channels are specified, then there are implicit signal routes/channels to all possible destination, in the block/system where signal routes/channels are omitted. There is still no possibility for a channel/signal route to go from the environment to the environment, which is forbidden in SDL. During the algorithm, we must respect the following constraints.

- The output's signal name must exist in every **with** clause of the passed gates, signal routes and channels for the direction in which the signal is traveling. If no with clause is specified then the signal is allowed to pass any signal route, channel or gate.
- The optional **via** constraint specifies a list of names which shall match with a name of a gate, a signal route or a channel, in the given order[6]. This is, when we encounter a situation where there are more than one possible continuation path and the first in the via list matches one of them, then we must use that and only that continuation. After this we have used the first via constraint which is then removed from the list for that continuation. If we reach a process set and the list isn't empty then that process set is not a possible receiver for the output.
- The optional **to** constraint specifies the name that a receiving process set must have. If we reach a process set that does not have the specified name then that process set is not a possible receiver for the output.

An outlined version how each level is searched is in Appendix E.

[5]We allow multiple channels to be connected to multiple signal routes, this feature belongs to SDL-92.
[6]At the time of writing, sdl2xta can only handle a single via name

## 5 Process conversion

In this section, a solution to the problem of converting an $\mathcal{SDL}_{xta}$ process to an Uppaal process, is presented. There are two distinct areas in an $\mathcal{SDL}_{xta}$ process, namely the declaration area, and the process body.

### 5.1 Process' declarations

A process' declarations consists of both definitions and declarations.

**Gate definitions** If the process defined as a type based process then it might include gate definitions. In that case all gate definitions must come before any other definition. There is no output generated that corresponds directly to gates. In the conversion program gates are used to restrict the possible receivers of output signals. This is done in the signal analysis.

**Signal definitions** Signals internal to the process can be defined. A signal defined inside a process (type) definition can not be sent to another process (not even if the receiving process has the same type). The number of parameter must be the same as when the signal is used. Each instance of a signal definition, generates a unique Uppaal constant with the signal's name as identifier.

**Timer definitions** In $\mathcal{SDL}_{xta}$ timers cannot have any parameters which makes the timer definition a pure declaration. An Uppaal instance of the template timer shown in Figure 31 will be generated for each instance of the process definition. The timer has a variable set_time that is set to the duration for the timer when it is used. This value is initiated with zero if the timer is not assigned a value at its definition.

**Pid declarations** $\mathcal{SDL}_{xta}$ defines a datatype **pid** that corresponds to SDL's PId type. Pid is the only datatype that can be used to store process ids. The reason for this is the reuse of process id (see Section 4.3). All pid storages are global in the run-time system. The convertion program makes a slot in the global `pids[]` array. This is of no concern to the $\mathcal{SDL}_{xta}$ writer.

**Xta declarations** $\mathcal{SDL}_{xta}$ provides all variables in Uppaal's xta-format to be declared. All declarations will be local to the Uppaal process implementing the SDL-process in the run-time system. See Appendix C.

### 5.2 Process body

A process body contains **state** definitions. The **state**s have **input** definitions that are followed by *transition*s. A state can also have **save** constructs. *Transition*s are built up by *action statement*s and/or *terminating statement*s.

## 5.3 States, Inputs and Queue interaction

**SDL's SDL-process and queue interaction.** In SDL control communication is done by events. The outlined schema is slightly simplified.

The SDL-process communicates with its queue with a *Next-Signal* event. As argument to the *Next-Signal* event the SDL-process sends the *save-set* of its current state. The *save-set* is the set of signals the process not shall handle until it has changed its state.

The queue now communicates with the SDL-process using a *Input-Signal* event. In this event the queue passes the next signal on the queue that is not an element of the *save-set*. If the queue is empty or if the queue only holds signals included in the *save-set*, then this communication must wait until a new signal arrive to the queue.

After receiving the *Input-Signal* event the SDL-process directs its flow of control to the transition corresponding to the <state, signal> pair. If no such pair exists or if the condition to take the transition is not fulfilled then the signal is discarded.

**The SDL-process and queue interaction translated to UPPAAL.** In our implementation we strive to make the queue as simple as possible. Our queue is generic: all instances use the same UPPAAL template, parameterized for that instance. We have chosen to let the SDL-process make all the decisions by itself, treating the queue as a passive object. For each state we let the SDL-process synchronize with its queue with a *nextsig*!. When the queue performs the associate *nextsig*? it also puts the next signal in a global variable (that is only used in this communication). In the next step, the SDL-process decides what to do with the signal. Until now nothing has happened in the queue.

- If there is an **input** construct for this <state, signal> pair and the construct either has no enabling condition or the enabling condition is true, then the SDL-process synchronizes with the queue with an *accceptsig*!, copying the **sender** and optional parameters for the signal. When the queue gets the corresponding *acceptsig*? it knows that the signal is regarded as consumed and moves the untested signals forward in its local queue, filling the gap of the consumed signal. The SDL-process now enters the selected transition.

- If there is a **save** construct for this <state, signal> pair, then the SDL-process and queue synchronize with a *savesig*. At the *savesig*!, the SDL-process goes directly back to the state where it came from, ready to do a new *nextsig*!. The queue moves its pointer to the next signal in the queue.

- If the signal is neither accepted nor saved then the signal must be discarded. This is also done with an implicit input transition where the SDL-process moves back to the same state again ready for a new *nextsig*!. In that move it synchronizes with the queue with an *acceptsig*! telling the queue to remove the signal as if it was consumed.

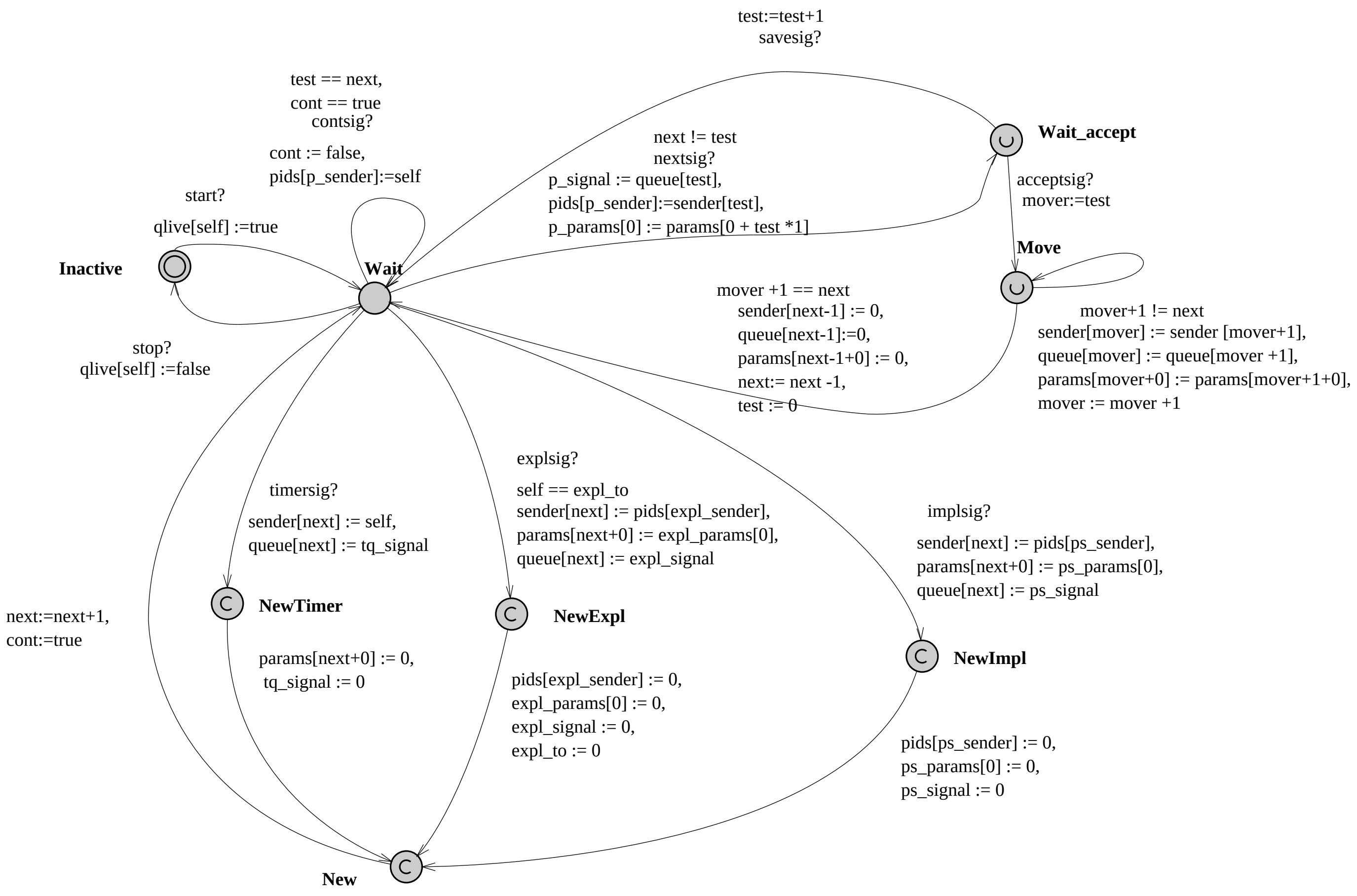


Figure 28: The *Queue* UPPAAL process generated for one parameter.

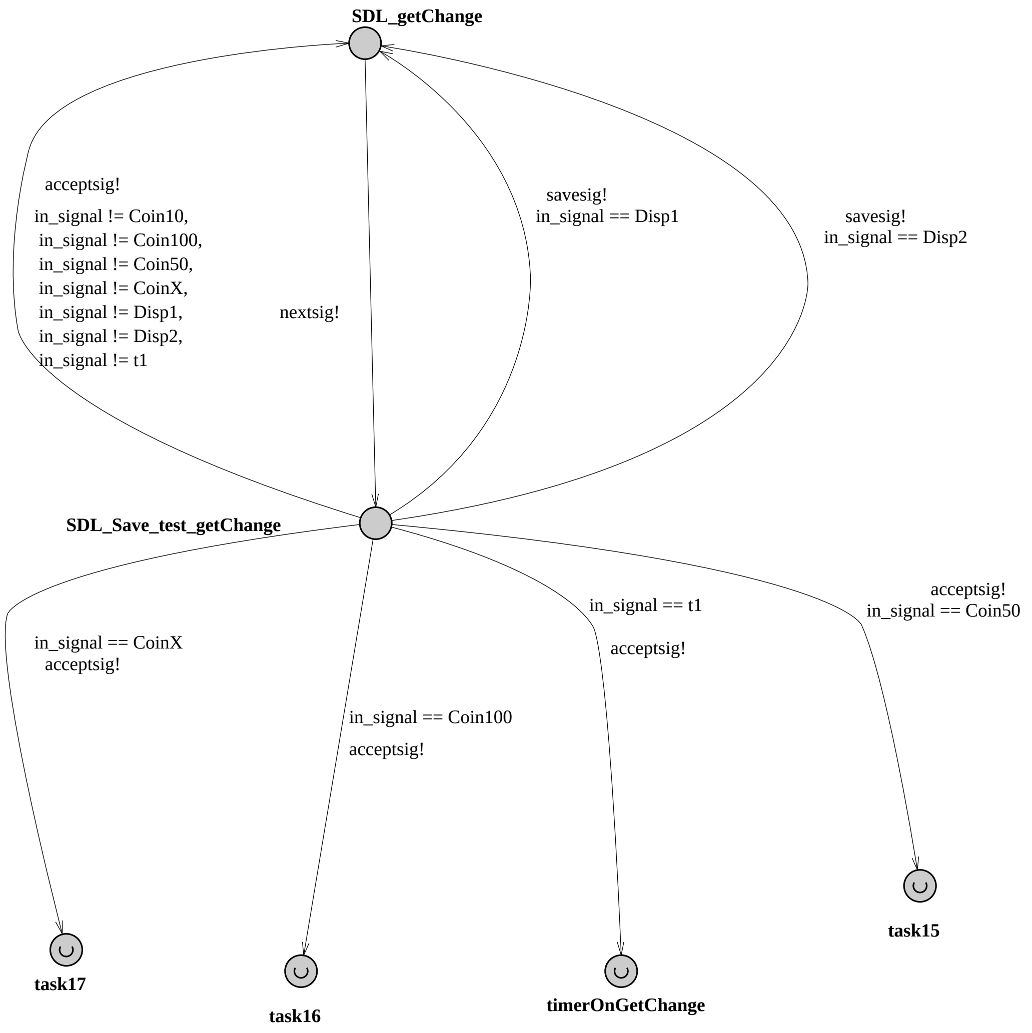


Figure 29: Excerpt of input in UPPAAL.

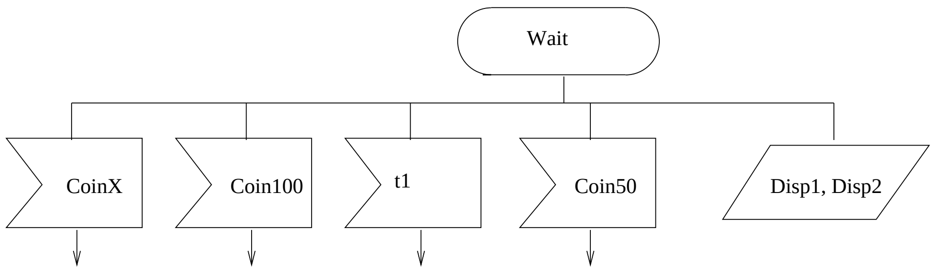


Figure 30: Excerpt of input in SDL.

## 5.4 Control Structure Translate

**Translation.** Because of the existing *shortcuts* in the **state** and **input** area there is no one-one mapping between a declared <state, input> pair and a transition. There are a few pitfalls:

- State names can appear in multiple state constructs. A state's input is the sum of the inputs where the state name appears.
- Asterisks can be used in the **state** construct. This means that the inputs of that state construct is for all of the SDL-process states. It is also possible to declare a list of states that shall not be concerned by the state construct.
- Multiple states can be declared in a state construct. This means that the inputs to that state construct is for the mentioned state names.
- Asterisks can be used in the input construct meaning that all signals, except those declared elsewhere for that state name, take the transition that follows in the construction.
- Asterisks can also be used in the save construct, meaning that all signals, except those declared elsewhere for that state name, are included in the *save-set*.

For each state we generate:

```
<state> -> <state_test> {
  sync nextsig!;
}
```

Now the queue has put the signal to test in `in_signal` and the parameters in `in_params[0..N-1]` for N parameters. The **sender** has been set to the sender pid of the signal, of course in-directed to the `pids` array. Now we set our `doNotDiscard`-set to $\emptyset$.

For every input signal in the state we generate:

```
<state_test> -> <first_state_of_transition> {
  guard in_signal == <actual_signal>
    [,<guard_on_input>];  // inside [] optional
  sync acceptsig!;
  assign  <input_par#_var> := in_params[ <input_par#> ];  // for every param #
}
```

when generating this we also add the signal to the `doNotDiscard`-set.
For every save signal in the state we generate:

```
<state_test> -> <state> {
  guard in_signal == <actual_signal>;
  sync savesig!;
}
```

when generating this we also add the signal to the `doNotDiscard`-set.
When we see an asterisk in an input or a save construct we remember it until we have gone through all constructs.

Finally we have three alternatives: discard, save, or run transition for non-mentioned signals. Discard means that we have neither a save nor an input asterisk construct for this state. In this case we accept the signal without doing any transition:

```
<state_test> -> <state> {
  guard in_signal != <signal>; // for all signals in doNotDiscard-set.
  sync acceptsig!;
}
```

If we have a save asterisk we generate:

```
<state_test> -> <state> {
     guard  in_signal != <signal>;  // for all signals in doNotDiscard-set.
     sync savesig!;
}
```

And last if we have an input asterisk

```
<state_test> -> <first_state_of_transition> {
  guard [<guard_on_input>,]  // inside [] optional
     in_signal != <signal>; // for all signals in doNotDiscard-set.
  sync acceptsig!; // no params in asterisk-input
}
```

## 5.5 Transitions

A transition is a sequence of *action*s optionally ending in a termination statement. The transition specifies the action(s) to be taken for an input in a state. Remember how states and inputs can be combined see 5.3. In case a transition not ending in a terminator statement (that is a nextstate, join or stop construct) the default is to return to the same state as before. What's complicating things is the decision statement. The decision statement is forking the execution flow in multiple paths, that possibly merge and continue after the end-decision. Each possible path is itself a transition. This means that following a transition, not ending in a terminator statement, we shall either continue after the last enddecision or return to the same state as before. The latter is complicated by the fact that a join can take an execution flow from one transition to another coming from a different state. This is why we have to keep track of our current state. We change the state when we encounter a nextstate construct. Our solution to the problem is to add a **join** to the enddecision if we are in a decision construct, else we add a **nextstate -**, meaning just go to same state. The **nextstate -** will generate a jump to the `state_choser` UPPAAL location. From there we go to our current SDL-state which we can decide from examine our `iSDL_State` variable. BNF for the transition can be found in Appendix B.12.

## 5.6 Best/Worst Case Execution Time

Best Case Execution Time (BCET) and Worst Case Execution Time (WCET) are not a part of SDL. They are added to $\mathcal{SDL}_{xta}$ to make the UPPAAL run-time system aware of the execution time. It is optional to use BCET/WCET in an $\mathcal{SDL}_{xta}$ specification. If they are not used UPPAAL can only verify the system re timers.

The syntax in $\mathcal{SDL}_{xta}$ for this extension is $[BCET, WCET]$, where BCET and WCET is a constant integer. The translation of this is done in three steps.

- At each incoming edge, the UPPAAL clock $c$ is assigned to zero.
- Add an invariant $c \leq WCET$ to the location. This will force the control to stay in the location maximally $WCET$ time units.
- Guard each outgoing edge with a $c \geq BCET$ condition. This will hinder the control to move from the location before $BCET$ time units have elapsed.

The translations of any $\mathcal{SDL}_{xta}$ actions will all include this addition if a BCET/WCET pair is specified.

## 5.7 Label

A label is nothing else than a named point in the execution path where it is possible to join branches. A label is not an SDL action. In the translation we generate an UPPAAL location for the label because it simplifies the translation and makes the generated UPPAAL code more easy to read. Whenever we encounter a "**join** *label*" action we simply add an edge to the *label* location. The code generated by a label is:

```
<label location>  -> <next location> {
}
```

The <*label location*> is urgent so it takes no execution time in UPPAAL. The BNF for the label construct can be found in Appendix B.13.

## 5.8 Output

An output is an SDL action to send signals to other processes. The output construct consists of an **output** keyword followed by a signal identifier with optional parameters inside parentheses. The parameters depending on how the signal is defined. The **to** and the **via** constructs can be used to constrain the possible receivers of the signal. Section 2.10 describes the SDL's output construct and Section 4.2 describes explicit and implicit signal delivery. Section 4.4 describes how outputs are analysed and how the possible receivers are determined.

**Explicit output.** With explicit output the pid of the receiving process instance is known. In this case the we generate a synchronization with the *Expl* process (explained in Figure 21 and Section 4) using the sdl_explicit channel.

```
<output location> -> <next location> {
  sync sdl_explicit!;
  assign expl_signal:= <signal>,
    expl_to := pids[<pid>],
    expl_params[#] := <param#>, // for all params
    pids[expl_sender] := self,
;
 }
```

The variables `expl_signal`, `expl_to`, `expl_params[]`, and `pids[]` are all global. The *Expl* process discards the signal if there is no process instance with pid `expl_to`.

**Implicit output.** With implicit output we do not know the pid of the receiving process. Happily all signal analysis is already done and the implementation is expressed in the PSO process for the process set. The synchronization is named after the output. The `via_<via>` and the `to_<to>` suffixes are optional. Each unique output has its own synchronization. The synchronizations used in the output constructs are parameters to the UPPAAL template for the SDL-process and the PSO process.

```
<output location> -> <next location> {
   sync output_<signal>_via_<via>_to_<to>!; // via and to optional
   assign  pso_signal:=<signal>,
     pso_params[#] := <param #>,
     pids[pso_sender] := self;
}
```

The variables `pso_signal`, `pso_params[]`, and `pids[]` are global, but in the synchronization the PSO process copies them and stores them locally.

## 5.9 Task

In $\mathcal{SDL}_{xta}$ a task is identical to an UPPAAL assign. This means that there is no need to convert anything in an $\mathcal{SDL}_{xta}$ task. There is one exception to this, namely the automated indirection of the **pid** type.

Generated xta code looks like:

```
<task location> -> <next location> {
  assign  <task goes here as is>;
}
```

## 5.10 Create and stop

The **create** and **stop** constructs is part of the *Dynamic handling of processes* concept 4.3. In this section we show what we generate for the SDL-process.

**SDL's create semantic.** A create constuct consists of a the **create** keyword and a identifier of a process set. The **create** construct creates a new process instance in the specified process set. It is not possible to start a process instance without any connection to the SDL structure. After creation the **offspring** variable will hold the pid of the newly started process or zero if no process has been started. The latter can happen when there is a maximum number of processes declared for the process set.

**Create in $\mathcal{SDL}_{xta}$.** In $\mathcal{SDL}_{xta}$ each create construct is analysed, so the process-set referred is known at generation time. The generated code for a create construct.

```
<create location> -> <next location> {
  sync <create ps sync>!;
  assign <ps offspring> := offspring,
    pids[<create ps parent>] := self;
}
```

We synchronize with the PSI process of the process set where the new process shall be created. This process are referred to as PSI_c for know. Here we do a little trick to transfer exchange pids between the creating and the created process instances. We assign the PSI_c's variable for offspring purpose with the created pids offspring variable. The offspring variable is an index to the global array `pids[]` where the process keeps its SDL **offspring** value. Now the PSI_c knows where to assign the pid of the created process instance. We also assigning the PSI_c's variable for parent transfer to the creating process pid, namely **self**. Notice that this variable, as it is a **pid** variable is located in the `pid[]` array.

**SDL's stop semantic.** In SDL **stop** is the end of the execution for the process instance. It dies and can never be reborn. All explicit signals for this pid will be discarded.

**Stop in $\mathcal{SDL}_{xta}$.** When stopping a process instance in $\mathcal{SDL}_{xta}$ we have some clean up to do due to the reuse of pids and UPPAAL processes. Section 4.3 describes the dynamic creation and deletion of processes in $\mathcal{SDL}_{xta}$. In Figure 26 shows the xta code of the clean up. The generate of each stop construct is

```
<stop location>  -> iSDL_Stop {
}
```

## 5.11 Decision

As discussed in the *Transition* Section 5.5, the decision construct is an important part of the flow control. In this section we focus on decision itself. There is a variety of valid decision constructs. We show how the question and answer parts can be combined in a semantically correct way. The decision is built up of a question (the head) and answers (the body).

**Semantic rules for decision. Rules:**

- A question must have at least two answers.
- Answers must be mutually exclusive and should not overlap.
- One, and only one, answer may be an **else** answer, which is the complement of all other answers.
- The question and the answers or range conditions of the answers must either be of the same sort or informal text.
- If the question is of **any** type, then all answers must be empty and no **else** answer can be used.

**Decision using any question.** The **any** question can be used with empty answers. It is a way to express non-determinism in SDL. It look like this:

```
decision any;
(): <transition 1>
(): <transition 2>
...
(): <transition N>
enddecision;
```

Every transition must have equal chance to be taken. This case is very simple to generate in UPPAAL. We simply omit the guards for all answers. UPPAAL's model checker will check all possible continuations. The translation of the **any** variant is:

```
<decision location> -> <transition n> {
}
```

**Decision using boolean question.** A boolean question makes the decision into an if statement. The answers to a boolean question can only be **true** or **false**. In SDL true and false are of the predefined type *bool*. In $\mathcal{SDL}_{xta}$ true and false can only be used in decision answers for boolean questions. The boolean (value) type is usually represented in UPPAAL as an integer value with range 0..1 denoted `int[0,1]`. $\mathcal{SDL}_{xta}$ predefines true and false as 1 and 0 respectively. The boolean question is an $<$*IGuard*$>_{ta}$ or a variable of type `int[0,1]`. The latter will be transformed to *vaiable* == 1. Two examples of a boolean decision:

```
decision x > 23;
(true):
  <true transition>
(false):
  <false transition>
enddecision;
decision y;        // y is boolean - int[0,1]
(true):
  <true transition>
else:
  <y is not true transition>
enddecision;
```

The translation is always on the form $expr_1$ $rel$ $expr_2$ where $rel$ includes $!=$. When the answer is **true** we have the guard $expr_1$ $rel$ $expr_2$, obviously. For the **false** answer we do the inverted relation $expr_1$ $rel_{inv}$ $expr_2$. Where $rel_{inv}$ is defined as:

| | | |
|---|---|---|
| $<$ | $\rightarrow$ | $>=$ |
| $>$ | $\rightarrow$ | $<=$ |
| $<=$ | $\rightarrow$ | $>$ |
| $>=$ | $\rightarrow$ | $<$ |
| $==$ | $\rightarrow$ | $!=$ |
| $!=$ | $\rightarrow$ | $==$ |

The generate for a boolean decision is:

```
<decision location> -> <transition true> {
  guard <Expr1> <rel> <Expr2>; // true statement
}
<decision location> -> <transition false> {
  guard <Expr1> <relinv> <Expr2>; // true statement
}
```

This is not the simplest way the false statement, but the most tasteful. It is of course possible to generate the guard $(< expr >?1 : 0) == 0$ as we do in the else case.

**Decision using value question.** When the question is an $<IExpr>_{ta}$, each answer contains one or more value ranges. Logical **or** cannot be expressed in UPPAAL in other ways than with multiple transitions. We do a little trick here. Again we map the range conditions to 1 or 0 depending on whether the condition is true. Now by summing up the values and checking against 0 we know if the branch should be taken or not.

First we show how each type of <*range*> is transformed. For question $<IExpr_1>_{ta}$ the answers are transformed as follow:

| | | |
|---|---|---|
| $<rel>$ $<IExpr_2>_{ta}$ | $\rightarrow$ | $(<IExpr_1>_{ta}$ $<rel>$ $<IExpr_2>_{ta}?1{:}0)$ |
| $<IExpr_2>_{ta}$ | $\rightarrow$ | $(<IExpr_1>_{ta} == <IExpr_2>_{ta}?\ 1\ :\ 0)$ |
| $<IExpr_{2_1}>_{ta}{:}<IExpr_{2_2}>_{ta}$ | $\rightarrow$ | $(<IExpr_1>_{ta} >= <IExpr_{2_1}>_{ta}?\ 1\ :\ 0)$ |
| | | $*\ (<IExpr_1>_{ta} <= <IExpr_{2_2}>_{ta}?\ 1\ :\ 0)$ |

For the example

```
decision <Expr1> ;
(2,4:8,>10):
   <transition 1>
(<2):
   <transition 2>
(else):
   <else transition>
enddecision;
```

We generate:

```
<decision location> -> <transition 1> {
  guard (<Expr1> == 2 ? 1 : 0)
  + ((<Expr1> >= 4 ? 1 : 0)*(<Expr1> <= 8 ? 1 : 0))
  + (<Expr1> > 10 ? 1 : 0) != 0;
}
<decision location> -> <transition 2> {
  guard (<Expr1> <2 ? 1 : 0) != 0;
}
<decision location> -> <else transition> {
  guard (<Expr1> == 2 ? 1 : 0)
  + ((<Expr1> >= 4 ? 1 : 0)*(<Expr1> <= 8 ? 1 : 0))
  + (<Expr1> > 10 ? 1 : 0)
  + (<Expr1> <2 ? 1 : 0) == 0;
}
```

$\mathcal{SDL}_{xta}$ does not check whether ranges overlap. If they do, the decision becomes non-deterministic. This solution guarantees that the **else** branch is not taken when an other is possible.

## 5.12 Set/reset of timers

SDL timers are discussed in Section 2.12. In $\mathcal{SDL}_{xta}$ a timer a always set to expire a duration from now. The $\mathcal{SDL}_{xta}$ specification writer must not include the SDL keyword **now** as an argument to set. Both the set construct and the reset construct generates a synchronization to the timer with the name given as parameters to the construct.

**Set.** For a set we generate:

```
<set location> -> <next location> {
  sync <timer>_set!;
  assign <timer>_set_time:= <set expr>;
}
```

If the set is specified with only the timer as argument, then the `<timer>_set_time` will not be assign anything at this point but at start up of the SDL-process.

**Reset.** For a reset we generate:

```
<reset location> -> <next location> {
  sync <timer>_set!;
}
```

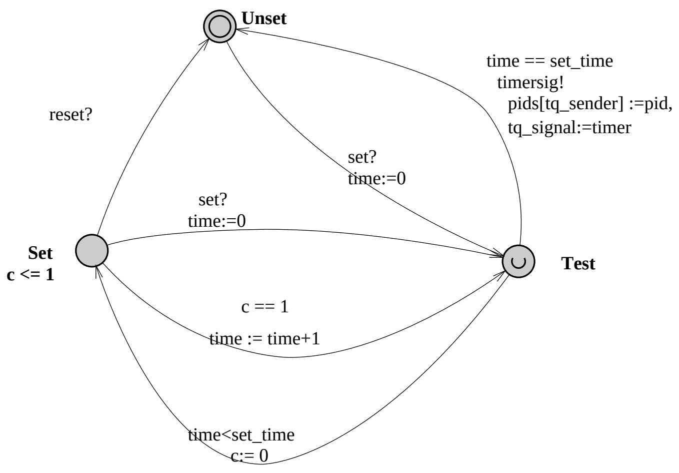


Figure 31: The *Timer* Uppaal process.

## 5.13 Nextstate

The nextstate construct is a way for the specification writer to change state of the SDL-process. The translation of this is simply to move the control to the Uppaal location representing that SDL state. For the cases where a transition not ending in a *termination statement* and for cases where we shall go to the same state without knowing (by the transition) what state we came from. We must have edges to all possible SDL states. We have chosen to make one such Uppaal location called `iSDL_State_choser` and we use a variable iSDL_State to keep track of next state. This makes it possible to always go to the iSDL_State_choser location whenever a transition is done. If no nextstate ends the transition, then we still know were to go by the variable iSDL_State.

The generated code for a nextstate construct.

```
<nextstate location> -> iSDL_State_choser {
  assign iSDL_State := <State>_Val; \\ not when ``nextstate -''
}
```

## 5.14 Join

As join has the function of goto in ordinary programming languages and we generate a loction with name *label* when we encounter a label. The translation of join is simple.

The generated code for a nextstate construct.

```
<join location> -> <label> {
}
```

## 6 Brief example

In this section we make a brief example of an $\mathcal{SDL}_{xta}$ system to show how the program works in practice. The full $\mathcal{SDL}_{xta}$-code can be seen in Appendix F.

### 6.1 $\mathcal{SDL}_{xta}$ representation

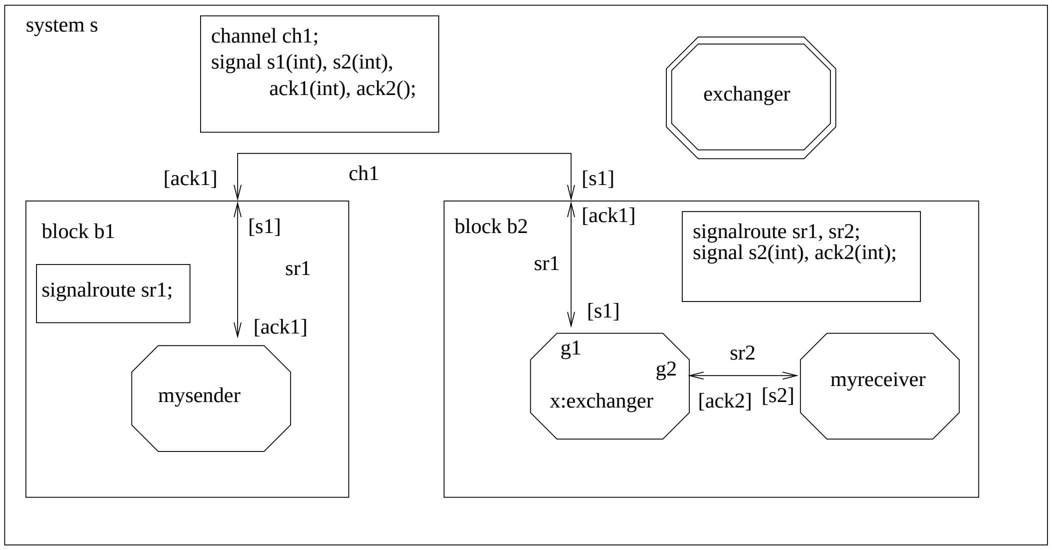


Figure 32: The SDL system.

**System** $s$ In Figure 32 we can see the system's highest abstraction level. We have the system $s$ which has no connection to the environment as $\mathcal{SDL}_{xta}$ forbids such connection. The system has two blocks $b1$ and $b2$, which are connected with the channel $ch1$. Channel $ch1$ conveys signal $s1$ from $b1$ to $b2$, and signal $ack1$ from $b2$ to $b1$. There are definitions for signals $s1$, $s2$, $ack1$, and $ack2$ at the system level. There is also a definition for channel $ch1$ and a definition for process type $exchanger$. Figure 32 shows also the inside of block $b1$ and $b2$.

**Block** $b1$ Inside block $b1$ there is a process $mysender$ and a signal route $sr1$ defined. The signal route connects the process $mysender$ and the outer channel $ch1$. Signal route $sr1$ conveys signal $ack1$ to the $mysender$ process from the block's environment and signal $s1$ in the opposite direction. Signal route $sr1$'s behaviour matches the behaviour of channel $ch1$ which is a must in this case, because no other signal route is connected to the channel.

**Block** $b2$ Inside block $b2$ there are two processes $mysender$ and $x$, where process $x$ is declared based on the type $exchanger$. There are also two signal routes defined $sr1$ and

$sr2$. Signal route $sr1$ conveys signal $s1$ from the environment, where it is connected to channel $ch1$, to process $x$ via gate $g1$, and also signal $ack1$ in the opposite direction. Signal route $sr2$ conveys signal $s2$ from process $x$ via gate $g2$ to the process $myreceiver$, and also signal $ack2$ in the opposite direction.

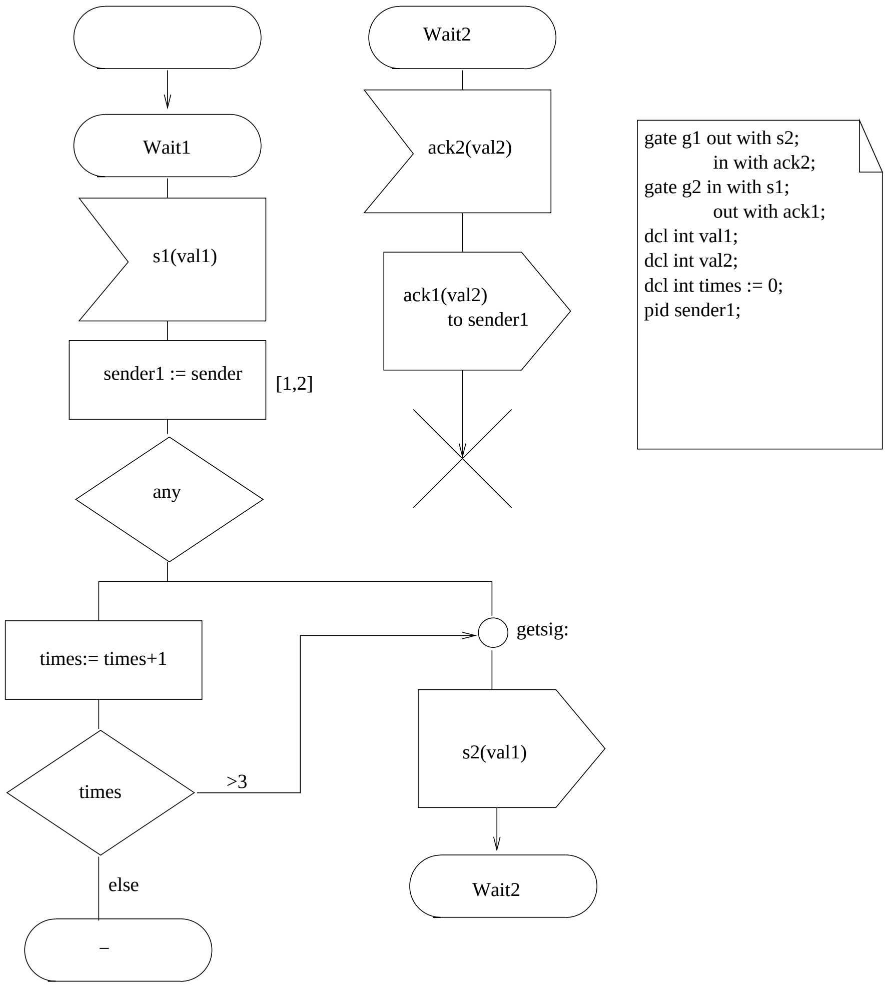


Figure 33: The xchanger process type.

**Process type** *exchanger* In Figure 33 we can see the SDL state chart of the process *type exchanger*. Process type *exchanger* has two states $Wait1$ and $Wait2$. When it starts it goes to state $Wait1$.

At $Wait1$ it ignores all inputs besides signal $s1$. When it got an input $s1$ with argument $val1$ it performs a task which takes one to two time units and saves the current **sender** value to remember the pid of the sender. At the decision it ramdomly selects one of two paths. In the first path the variable *times* is incremented and then a test is done if *times* is greater then three. If $times$ is greater then three it joins at label $getsig$, else it waits for next signal in the same state. The second path sends the signal $s2$ with the argument $val1$ as it just received and change state to $Wait2$.

At $Wait2$ it ignores all inputs besides signal $ack2$. After receiving an $ack2$ signal it sends an $ack1$ signal to the process with pid $sender1$ as it saved earlier. The argument $val2$ is passed from signal $ack2$ to signal $ack1$.

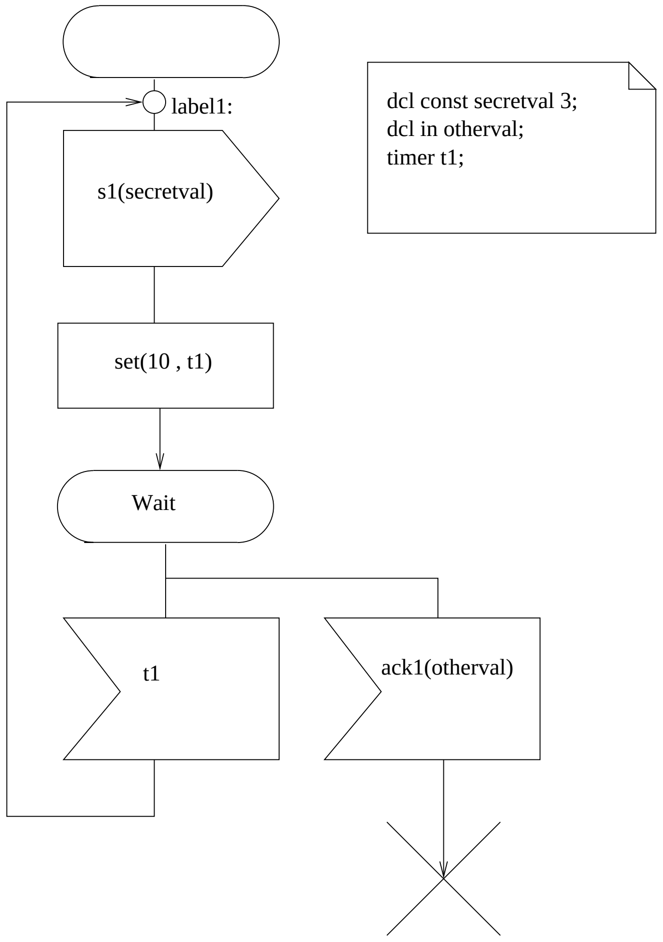


Figure 34: The SDL sender process.

**Process** $mysender$ In Figure 34 we can see the SDL state chart of the process $mysender$. The process $mysender$ has only one state, namely $Wait$. It starts with a label called $label1$ (not shown in the figure), then it sends a signal $s1$ with $secretval$ as argument, sets its $t1$ timer so it expires ten time units from now and lastly goes to state $Wait$. This is the initialization of the process. At state $Wait$ it can receive a signal $ack1$, then it is pleased and stops. If the two time units elapses before any $ack1$ signal has arrived, then the timer expires and the process joins $label1$ and does the initialization again.

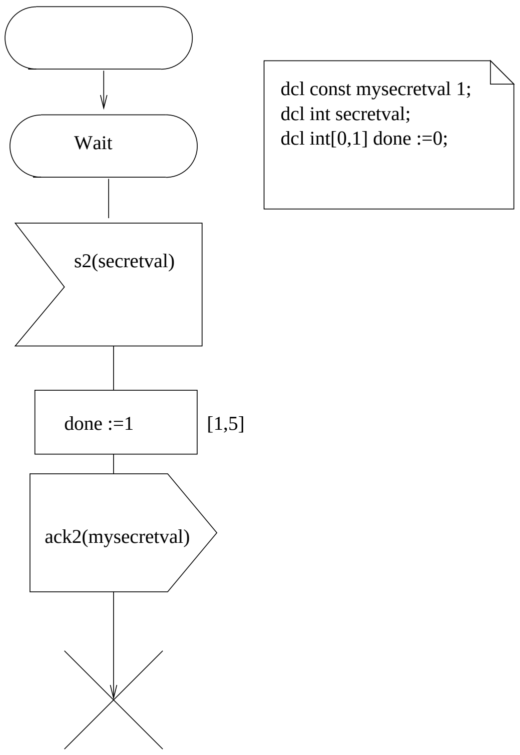


Figure 35: The SDL receiver process.

**Process** $myreceiver$ In Figure 35 we can see the SDL state chart of the process $myreceiver$. The process $myreceiver$ has one state called $Wait$. At startup it goes directly to the $Wait$ state. When it receives an $s2$ signal it performs a task that takes one to five time units and assigns one to the variable *done*. The process continues and sends out an $ack2$ signal, futher it stops its execution.

## 6.2 The UPPAAL result

In this section we present the UPPAAL run-time SDL-processes for the SDL processes introduced in the previous section. The full output can be seen in Appendix G.

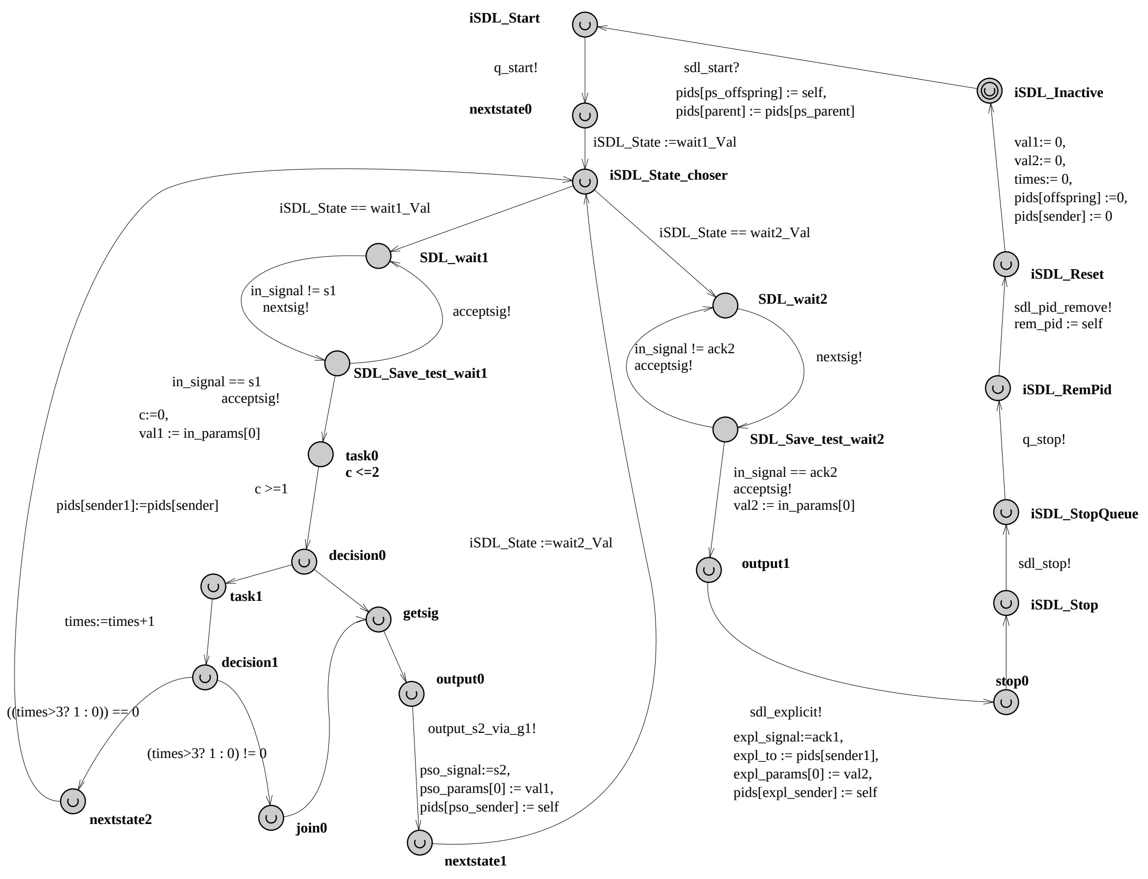


Figure 36: The exchanger UPPAAL process.

**The *exchanger* process in UPPAAL.** In Figure 36 we can see the result of the translation of the *myreceiver* process. The *exchanger* process includes three examples of constructs not seen in the earlier processes. Firstly, at location *task*0 a task is performed with a pid value. The task is usually a simple assign, but as ths is a pid assignment it is done to the `pids[]` array with the assigned variable as index. Secondly, at location *decision*0 an any-decision is performed. The outgoing answers are always empty for an any question. Thirdly, at location *output*1 an explicit output is preformed. The addressed process is the process with pid equal to the value of the *sender*1 variable.

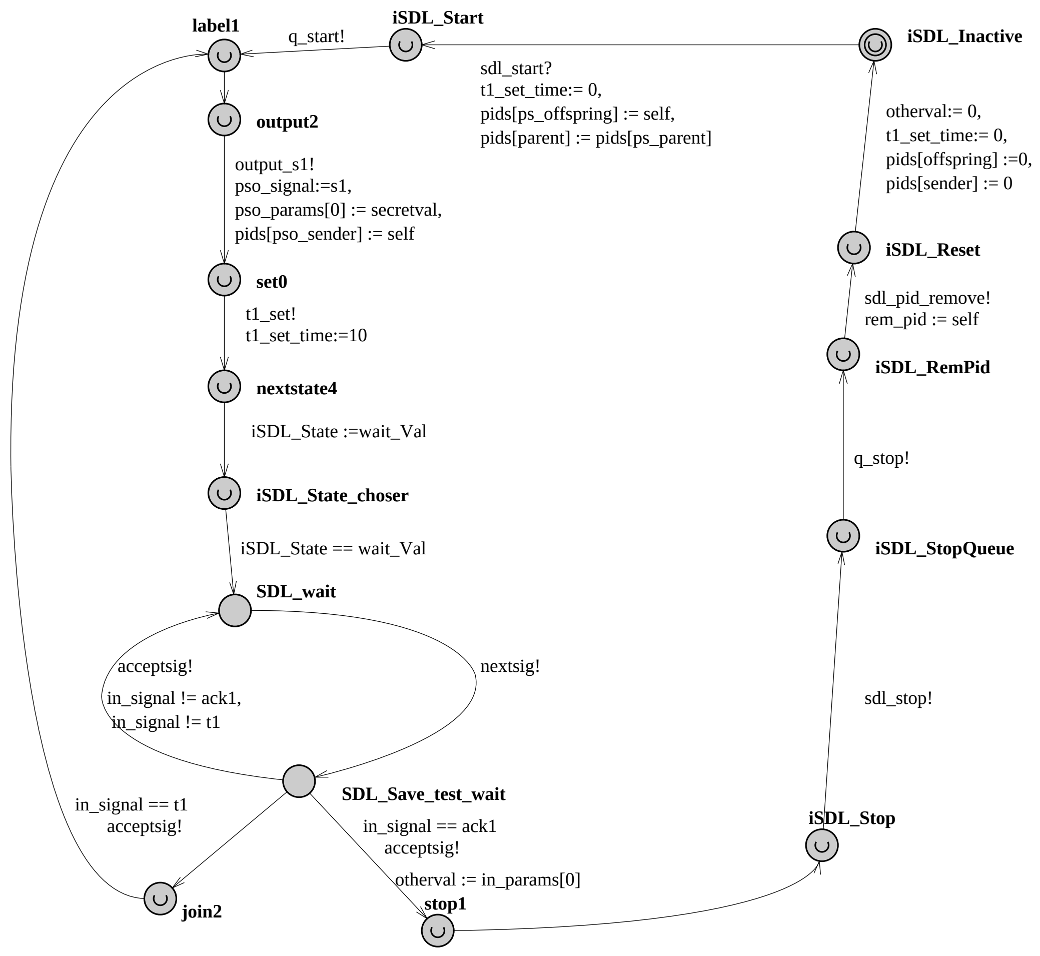


Figure 37: The sender UPPAAL process.

**The *mysender* process in UPPAAL.** In Figure 37 we can see the result of the translation of the *mysender* process. After activation the SDL initialization begins at location $iSDL\_Start$. We recognize the label $label1$, the output for signal $s1$, timer $t1$ that is set to two time units, and the nextstate $Wait$. At nextstate the variable $iSDL\_state$ is set to a value representing state $Wait$. At the location $iSDL\_state$ it can only go to the $Wait$ state. At state $Wait$ the process can accept signal $s1$ and timer $t1$ as input. If timer $t1$ expires, then the process comes to a join and goes to label1. If the process gets a signal $ack1$, then the process comes to a stop location and goes from there to the $iSDL\_Stop$ location wherefrom clean up begins.

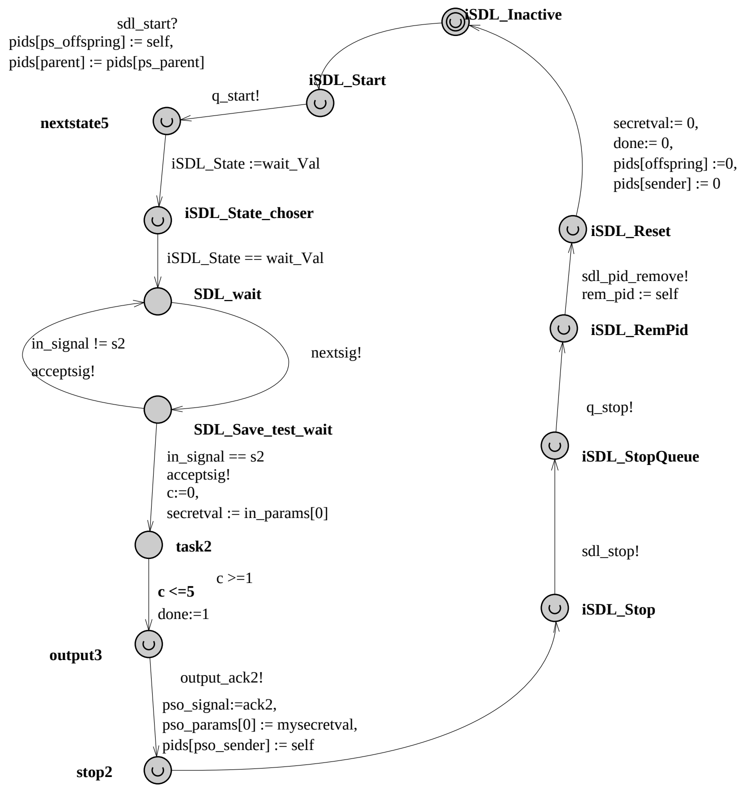


Figure 38: The receiver UPPAAL process.

**The** ***myreceiver*** **process in UPPAAL.** In Figure 38 we can see the result of the translation of the *myreceiver* process. After activation the process goes directly to the state $Wait$. This is done by the procedure to assign the $iSDL\_state$ the value that represents the $Wait$ state and then go to the $iSDL\_State\_choser$ location from where it must go to the location representing the state $Wait$.

At state $Wait$, the process waits until it receives an $s2$ signal, then it outputs an $ack2$ signal. Notice that the $s2$ signal's parameter assigns the variable *secretval* and that the $ack2$ signal is sent with the variable *mysecretval* as parameter. After the output the process stops.

## 6.3 Using the verifier in UPPAAL

In UPPAAL there is a verifier where we can prove properties.

There are a few possibilities for proving the properties that we want the exchanger example to have. Because all of the three processes mysender, myreceiver, and x are determined to stop. We can prove that starting one leading to stop. We give the verifier, for each process the query:

```
<process>.iSDL_Start --> <process>.iSDL_Stop
```

Another possibility is to prove that a process stops is to modify it so it assigns a global variable[7] when it is finished. For example if we assign <process>_done the value 1 at the end of the process:

```
A<> <process1>_done == 1 and <process2>_done == 1 (etc..)
```

A<> means that there is always a possible path that leads to the property, i.e., in any possible state, if we have no live locks, the property will hold in the future.

To prove that the example doing its work, we make *mysender*'s value *otherval* and *myreceiver*'s value *secretval* global. Then we prove:

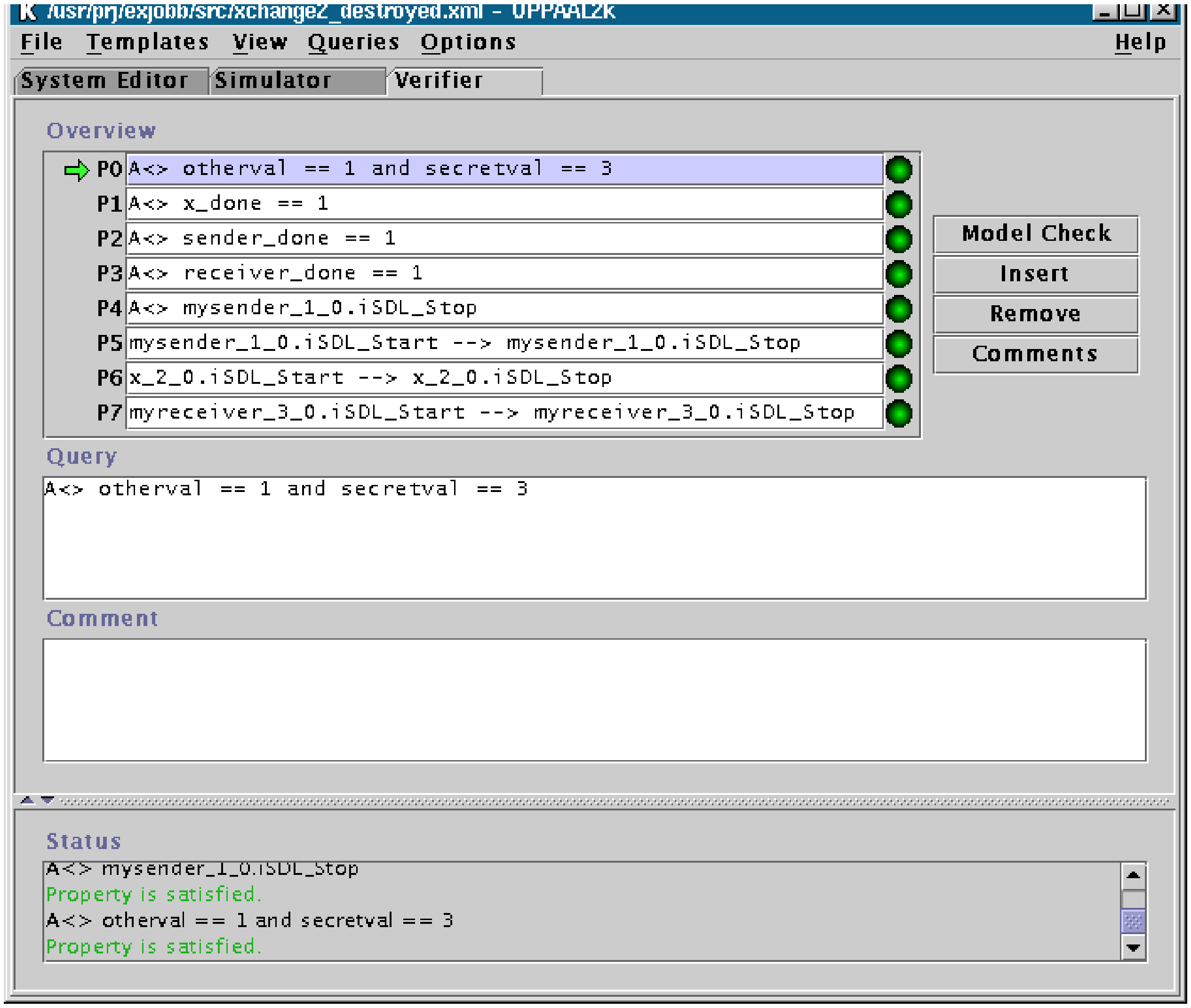
```
A<> otherval == 1 and secretval == 3
```

This proves that the prcesses alway sometime in the future will have exchanged their values.

To make the verifier go (much) faster it is recommended to remove all locations after the location *iSDL_Stop* in the processes whenever there is no dynamic creation of processes. In next version this will be done by the sdl2xta program.

Figure 39 shows what it looks like in UPPAAL's verifier.

[7]We cannot write queries in UPPAAL for local variables.



```
P0 A<> otherval == 1 and secretval == 3
P1 A<> x_done == 1
P2 A<> sender_done == 1
P3 A<> receiver_done == 1
P4 A<> mysender_1_0.iSDL_Stop
P5 mysender_1_0.iSDL_Start --> mysender_1_0.iSDL_Stop
P6 x_2_0.iSDL_Start --> x_2_0.iSDL_Stop
P7 myreceiver_3_0.iSDL_Start --> myreceiver_3_0.iSDL_Stop
```

```
A<> otherval == 1 and secretval == 3
```

```
A<> mysender_1_0.iSDL_Stop
Property is satisfied.
A<> otherval == 1 and secretval == 3
Property is satisfied.
```

Figure 39: The Uppaal verifier.

# 7 Conclusion

## 7.1 Related work

- The IF[IF] project at Verimag, SDL specifications undergo several translations. The resulting PROMELA representations is used with the SPIN model checker.
- Telelogic Tau SDL Suite is a commercial tool for graphically building of SDL system and simulation of live performance [Tau].
- The SVE (System Verification System) tool developed at Siemens[Siemens]. Relied on using BDD-based symbolic model checker.
- ATMTS, Algebraic Tools for Modelling Telecommunication Systems. Steggles' group at University of Newcastle UK, jointly with LMU Munich, Germany have done a Formal Model for SDL Specifications based on Timed Rewriting Logic [ATMTS].

## 7.2 Future work

**Configuration file.** There should be a configuration file where the size of the queues and maximum number of parameters can be set.

**Default time consumption.** An idea is to assign default time consumption per $\mathcal{SDL}_{xta}$ construct. The user would only have to write exceptions.

**Take care of buffer overflow.** Buffer overflow in the queue should be detected. Today we have no such detection.

**Procedures and services.** The procedure level is not included in $\mathcal{SDL}_{xta}$. It it possible to expand $\mathcal{SDL}_{xta}$ to do local procedure calls in a UPPAAL automaton using a stack for the return "address". The problem here is the same as with the queues, we can neither allow unbounded nor dynamic size.

**Delayed channels.** It would be nice to have delayed channels, where delay could be in a "flexible span". This will unfortunately force us to only write fully specified systems. The way we implement dynamic implicit behaviour will be impossible.

## 7.3 Summary

In this work we have described how it is possible to convert an SDL syntax to UPPAAL preserving the characteristics of the environment an SDL process work in. We have also added a way to express time comsumption for constructs in $\mathcal{SDL}_{xta}$ processes in order to do an analysis of an SDL system, with both timers and execution time.

## References

[AD94] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183-236, April 1994.

[ATMTS] L. J. Steggles an P. Kosiuczenko *A Formal Model for SDL Specifications based on Timed Rewriting Logic*, ¡br¿ In Proceedings of Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science, Vol. 15, 1998.

[ASN.1] ITU-T Recommendations X.280 and X.680-683: *The Abstract Syntax Notation One* (ASN.1)

[Brau84] W.Brauer: *Automatentheorie*, Teuber Verlag, 1984.

[EHS97] Jan Ellsberger, Dieter Hogrefe and Amardo Sarma: *SDL, Formal Object-oriented Language for Communication Systems* Prentice Hall 1997.

[Holtz91] G.Holtzmann: *Design and Validation of Computer Protocols*, Prentice-Hall International 1991.

[IF] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier: *IF: A validation environment for timed asynchronous systems*, Proceedings of CAV'2000, Lecture Notes in Computer Science 1855, pages 543-547. Springer Verlag, July 2000.

[ISD] ISD Datasystem AB: http://www.isd.se/

[LNCS2078] Lecture Notes in Computer Science; Vol. 2078 *SDL 2001: Meeting UML* 10th International SDL Forum, Copenhagen Denmark June 27-29, 2001, Proceedings, Springer–Verlag 2001.

[LPY97] Kim G. Larsen and Paul Pettersson and Wang Yi: UPPAAL *in a Nutshell*, Int. Journal on Software Tools for Technology Transfer, Springer–Verlag 1997.

[SDL02] SDL Forum 2002 at http://www.sdl-forum.org/SDL/

[Siemens] Franz Regensburger and Aenne Barnard: *Formal Verification of SDL systems at Siemens mobile phone department*, Bernard Steffen, editor, Proceedings of TACAS'1998, Lecture Notes in Computer Science 1384, pages 439-455. Springer Verlag, 1998.

[Tau] Telelogic Tau SDL suite. *www.telelogic.com*

[Z.100] ITU-T Recommendation Z.100: *Specification and Description Language Z.100 SDL*

## A Differences between SDL-92 and $\mathcal{SDL}_{xta}$

In this section we present the deviations in $\mathcal{SDL}_{xta}$ from SDL-92 and we suggest work-arounds, where appropriate. This list shall be considered as an overview and not as a complete deviation list. Appendix B and Appendix C defines $\mathcal{SDL}_{xta}$.

**Channels** Channels in $\mathcal{SDL}_{xta}$ cannot be **delayed** nor can any **substructure** be defined. This two deviations makes it possible for us to use our run-time algorithm for implicit communication between process sets described in Section 4.

**Types** We can only use the types that is included in UPPAAL. The **newtype**, **generator**, and **synonym** can not be used in $\mathcal{SDL}_{xta}$. Enumeration types can be simulated by doing a `const` declaration of each element and when the type is used, then used an int with range just enclosing the constant declared e.g. `int[1,3]` if the elements are declared from one to three.

**Virtual types** Virual types are not included in $\mathcal{SDL}_{xta}$, so none of the keywords **virtual**, **redefined**, or **finalized** can be used.

**Process context parameters** Process can not have parameters, so the **fpar** keyword cannot be used. The **fpar** keyword cannot be used in any other context either.

**Refinements** Refinements of any kind cannot be done, so the **atleast** keyword cannot be used in $\mathcal{SDL}_{xta}$.

**Inheritance** Inheritance is not included in $\mathcal{SDL}_{xta}$, so the keywords **inherits**, and **adding** are not used.

**Global time** In $\mathcal{SDL}_{xta}$ there is no absolut global time, all time is relative. Therefore do we not have any **now** in the system. When setting timers we set the duration rather than the exact time when the timer shall expire.

**Environment interaction** A **system** cannot communicate with its **env**ironment in $\mathcal{SDL}_{xta}$. This is a decision that has been made with consideration to UPPAAL. Our suggestion is to build the system in a substructure block and connect it to a simulated environment. This environment has to be specified and connected to the original system.

**Procedures** The **procedure** concept is not implemented in $\mathcal{SDL}_{xta}$. This is regarded as high priority future work.

**Packages** Packages is no implemented in $\mathcal{SDL}_{xta}$, so the **use** keyword cannot be used.

**Priority input** Priority input is no implemented in $\mathcal{SDL}_{xta}$, so the **priority** keyword connot be used.

**View** Variables in $\mathcal{SDL}_{xta}$ cannot be shared among processes in the way **view** works in SDL.

**Select** The SDL **select** construct is not implemented in $\mathcal{SDL}_{xta}$.

**Service** The SDL **service** construct is not implemented in $\mathcal{SDL}_{xta}$.

**Communication between a Process and its Queue** The exchange of control signals when a process communicates with its queue are not exactly the same in $\mathcal{SDL}_{xta}$ as in SDL. In Section 5.3 we describe our solution.

**Macrodefinition** Macros can not be defined in $\mathcal{SDL}_{xta}$.

**Added BCET and WCET** We have introduced the possibility to assign time-constraints on a process' actions. In Section 5.6 we describe this feature.

**System type** In $\mathcal{SDL}_{xta}$ the system specication cannot be type based.

**Multicast** The SDL multicast **via all** is not implemented.

**Continuous signals** Continuous signals are not implemented in $\mathcal{SDL}_{xta}$. They were planned to be implemented (see Queue), but the enabling conditions were hard to handle ... with the **provided** keyword.

**Enabling conditions**

**Spontaneous signals** Spontaneous signals, with keyword **none** are not implemented in $\mathcal{SDL}_{xta}$.

# B BNF of $\mathcal{SDL}_{xta}$

## B.1 BNF

In the Backus-Naur Form, a terminal is either indicated by not inclosing it within angle brackets (that is, the less-than sign and greater-than sign, <and>) or it is one of the two representations <name> or <character string>. Note that the two special terminals <name> or <character string> may also have semantics stressed and defined below.

The angle brackets and enclosed word(s) are either a non-terminal symbol or one of the two terminals <name> or <character string>. Syntactic categories are the non-terminals indicated by one or more words enclosed between angle brackets. For each non-terminal symbol, a production rule is given. For example

<block reference> *::=* **block** <block name> **referenced ;**

A production rule for a non-terminal symbol consists ot the non-terminal symbol at the left-hand side of the symbol ::=, and one or more constructs, consisting of non-terminal and/or terminal symbol(s) at the right-hand side. For example, <*block reference*> and <*block name*> in the example above are non-terminals; **block**, **referenced** and **;** are terminal symbols.

Sometimes the symbol includes an underlined part. This underlined part stresses a semantic aspect of that symbol, e.g. <*block name*> is syntactically identical to name, but semantically it requires the name to be a block name.

At the right-hand side of the ::= symbol several alternative productions for the non-terminal can be given, separated by vertical bars (|). For example,

```
<referenced definition>  ::=  <block definition>
                              | <block type definition>
                              | <process definition>
                              | <process type definition>
                              | <block substructure definition>
```

expresses that a <*referenced definition*> is either a <*block type definition*>, a <*process definition*>, a <*process type definition*> or a <*block substructure definition*>.

Syntactic elements may be grouped together by using curly brackets ({ and }). A curly bracketed group may contain one or more vertical bars, indicating alternative syntactic elements. For example,

<block interaction area> *::=* { <block area> | <channel definition area> }+

Repetition of curly bracketed groups is indicated by an asterisk (∗) or plus sign (+). An asterisk indicates thet the group is optional and can be futher repeated any number of times; a plus sign indicates that the group must be present and can further repeated any number of times. The example above expresses that a <*block interaction area*> contains at least one <*block area*> or <*channel definition area*> and may contain more <*block area*>s and <*channel definition area*>s.

If syntactic elements are grouped using square brackets([ and ]), then the group is optional. For example,

<valid input signal set> *::=* **signalset** [<signal list>] **;**

expresses that a <*valid input signal set*> may, but need not, contain <*signal list*>.

## B.2 Global

| | | |
|---|---|---|
| <sdl specification> | ::= | <GDecl>$_{ta}$ |
| | | <system definition> { <referenced definition> }$^*$ |
| <referenced definition> | ::= | <block definition> |
| | | \| <block type definition> |
| | | \| <process definition> |
| | | \| <process type definition> |
| | | \| <block substructure definition> |

## B.3 Signal definition

| | | |
|---|---|---|
| <signal definition> | ::= | **signal** <signal definition item> { ,<signal definition item> }$^*$ ; |
| <signal definition item> | ::= | <signal name> [<sort list>] |
| <sort list> | ::= | ( <param type>$_{ta}$ { ,<param type>$_{ta}$ }$^*$ ) |

## B.4 Communication syntax

| | | |
|---|---|---|
| <channel definition> | ::= | **channel** <channel name> [8] |
| | | <channel path> [<channel path>] [9] |
| | | **endchannel** [<channel name>] ; |
| <channel path> | ::= | **from** <channel endpoint> **to** <channel endpoint> **with** <signal list> ; |
| <channel endpoint> | ::= | { <block identifier> \| **env** } [**via** <gate identifier>] |

**Rule for channel gate.** <*gate identifier*> must be specified if and only if:

1. <*channel endpoint*> denotes a connection to a <*typebased block definition*> in which case the <*gate identifier*> must be defined directly in the block type for that block, or

2. **env** is specified and the channel is defined in a block substructure of a <*block type definition*> in which case the <*gate identifier*> must be defined in this block type.

| | | |
|---|---|---|
| <signal route definition> | ::= | **signalroute** <signal route name> |
| | | <signal route path> [<signal route path>] |
| <signal route path> | ::= | **from** <signal route endpoint> **to** <signal route endpoint> |
| | | **with** <signal list> ; |
| <signal route endpoint> | ::= | { <process identifier> \| **env** } [**via** <gate identifier>] |

**Rule for signal route gate.** <*gate identifier*> must be specified if and only if:

1. <*signal route endpoint*> denotes a connection to a <*typebased process definition*> in which case the <*gate identifier*> must be defined directly in the process type for that process, or

2. **env** is specified and the signal route is defined in a block type of a *<block type definition>* in which case the *<gate identifier>* must be defined in this block type.

## B.5 Referenced structures

| | | |
|---|---|---|
| <number of process instances> | ::= | **(** <NAT>$_{ta}$ [ **,** <NAT>$_{ta}$ ]**)** |
| <process reference> | ::= | **process** <process name> |
| | | [<number of process instances>] **referenced ;** |
| <process type reference> | ::= | **process type** <process type name> **referenced ;** |
| <block reference> | ::= | **block** <block name> **referenced ;** |
| <block type reference> | ::= | **block type** <block type name> **referenced ;** |
| <block substructure reference> | ::= | **substructure** <block substructure name> **referenced ;** |

## B.6 System definition

| | | |
|---|---|---|
| <system definition> | ::= | **system** <system name> **;** |
| | | { <entity in system> }+ |
| | | **endsystem** [<system name>] **;** |
| <entity in system> | ::= | <block definition> |
| | | \| <block reference> |
| | | \| <typebased block definition> |
| | | \| <channel definition> |
| | | \| <signal definition> |
| | | \| <signal list definition> |
| | | \| <block type reference> |
| | | \| <block type definition> |
| | | \| <process type definition> |
| | | \| <process type reference> |

**test** for *<channel definition>* see B.4.

## B.7 Block and block type definition

| | | |
|---|---|---|
| <block definition> | ::= | **block** <block name> ; |
| | | { <channel to route connection> \| <entity in block>}* |
| | | [<block substructure definition> |
| | | \| <block substructure reference>] |
| | | **endblock** [<block name> ] ; |
| <block type definition> | ::= | **block type** <block type name> ; |
| | | {<gate definition>}* |
| | | {<entity in block>}* |
| | | [<block substructure definition> |
| | | \| <block substructure reference>] |
| | | **endblock type** [<block name> ] ; |
| <typebased block definition> | ::= | **block** <typebased block heading> ; |
| <typebased block heading> | ::= | <block name> [<number of block instances>] **:** |
| | | <block type name> |
| <number of block instances> | ::= | **(** <NAT>$_{ta}$ **)** |
| <entity in block> | ::= | <signal definition> |
| | | \| <signal list definition> |
| | | \| <process definition> |
| | | \| <process reference> |
| | | \| <typebased process definition> |
| | | \| <signal route definition> |
| | | \| <process type definition> |
| | | \| <process type reference> |
| | | \| <block type definition> |
| | | \| <block type reference> |
| <channel to route connection> | ::= | **connect** <channel identifiers> |
| | | **and** <signal route identifiers> ; |
| <channel identifiers> | ::= | <channel identifier> {, <channel identifier> }* |

## B.8 Substructure

A substructure is like a system, it has channels that connect blocks. It is also the content of a partitioned block.

| | | |
|---|---|---|
| <block substructure definition> | ::= | **substructure** [<block substructure name>] ; |
| | | { <entity in system> \| <channel connection> }$^+$ |
| | | **endsubstructure** [<block substructure name>] ; |

if the <*block substructure name*> after the keyword **substructure** is omitted, it is the same as the name of the enclosing <*block definition*> or <*block type definition*>.

| | | |
|---|---|---|
| <channel connection> | ::= | **connect** <channel identifiers> **and** <subchannel identifiers> |
| <subchannel identifiers> | ::= | <channel identifiers> |

## B.9 Process and process types

```
<process definition>             ::= process <process name>[10] [<number of process instances>];
                                     { <entity in process> }*
                                     [<process body>][11]
                                     endprocess [<process name>] ;
<process body>                   ::= <start> { <state> }*
<typebased process definition>   ::= process <typebased process heading> ;
<typebased process heading>      ::= <process name> [ <number of process instances> ] :
                                     <process type name>
<process type definition>        ::= process type <process type name>;
                                     { <gate definition> }*
                                     { <entity in process> }*
                                     | <process body>[12]
                                     endprocess type [<process type name>] ;
<gate definition>                ::= gate <gate name> <gate constraint> ; [ <gate constraint> ;]
<gate constraint>                ::= { in | out } [13] [ with <signal list> ]
```

## B.10 Declarations in process

```
<entity in process>       ::= <signal definition>
                              | <signal list definition>[14]
                              | <variable definition>
                              | <timer definition>
<variable definition>     ::= { dcl <sdl decl>_ta
                              | pid <pid name> { , <pid name> }* ; }*
<timer definition>        ::= timer <timer definition item> { , <timer definition item> }* ;
<timer definition item>   ::= <timer name> [15] [ := <CExpr>_ta]
```

## B.11 States, Inputs and Queue interaction

```
<start>                ::= start; <transition>
<state>                ::= state <state list>;
                           { <input part>
                           | <save part>
                           | <spontaneous transition>
                           | <continuous signal> }* [16]
                           [endstate <state name>;]
<state list>           ::= {<state name>{,<state name>}*}
                           | <asterisk state list>
<asterisk state list>  ::= <asterisk> [( <state name> {, <state name>}* ) ]
<asterisk>             ::= *
```

| | | |
|---|---|---|
| <input part>[17] | ::= | **input** <input list>**;** [<enabling condition>] <transition> |
| <input list> | ::= | <asterisk input list> \| <stimulus>{ **,**<stimulus> }$^*$ |
| <asterisk input list> | ::= | <asterisk> |
| <stimulus> | ::= | {<signal identifier> \| <timer identifier>} |
| | | [**(** <$\mathcal{SDL}_{xta}$variable> {**,** <$\mathcal{SDL}_{xta}$variable> }$^*$ **)** ] |
| <save part> | ::= | **save** <save list>**;** |
| <save list> | ::= | { <signal list> |
| | | \| <asterisk save list> } |
| <asterisk save list> | ::= | <asterisk> |
| <spontaneous transition> | ::= | **input none;** [<enabling condition>] <transition> |
| <continuous signal> | ::= | **provided** <IGuard>$_{ta}$**;** <transition> |
| <enabling condition> | ::= | **provided** <IGuard>$_{ta}$**;** |

## B.12 Transition

| | | |
|---|---|---|
| <transition> | ::= | { <transition string> [<terminator statement>] } |
| | | \| <terminator statement> |
| <transition string> | ::= | { <action statement> }+ |
| <action statement> | ::= | [<label>] <action> [<$\mathcal{SDL}_{xta}$ execution time>]**;** |
| <$\mathcal{SDL}_{xta}$ execution time> | ::= | [ <BCET[18]> **,** <WCET[19]> ] |
| <action> | ::= | <task> |
| | | \| <output> |
| | | \| <create request> |
| | | \| <decision> |
| | | \| <set> |
| | | \| <reset> [20] |
| <terminator statement> | ::= | [<label>] <terminator>**;** |
| <terminator> | ::= | <nextstate> |
| | | \| <join> |
| | | \| <stop> [21] |

Refer to 5.6 for the BCET/WCET.

## B.13 Label

| | | |
|---|---|---|
| <label> | ::= | <connector name> **:** |

## B.14 Output

| | | |
|---|---|---|
| <output> | ::= | **output** <output body> |
| <output body> | ::= | <signal identifier> [<actual parameters> ] |
| | | { **,** <signal identifier> [<actual parameters> ] }* |
| | | [**to** <destination>] [**via** [22] <via path>] |
| <destination> | ::= | <pid expression> |
| | | \| <process identifier> |
| | | \| **this** |
| <pid expression> | ::= | **self** |
| | | \| **sender** |
| | | \| **parent** |
| | | \| **offspring** |
| | | \| <pid identifier>[23] |
| <via path> | ::= | <via path element> {**,**<via path element>}* |
| <via path element> | ::= | <signal route identifier> |
| | | \| <channel identifier> |
| | | \| <gate identifier> |

## B.15 Task

| | | |
|---|---|---|
| <task> | ::= | **task**<task body> |
| <task body> | ::= | <IAssign List>$_{ta}$ |
| | | \| <informal text> |
| <informal text> | ::= | **'** <charstring> **'** |

## B.16 Create and stop

| | | |
|---|---|---|
| <create request> | ::= | **create** <create body> |
| <create body> | ::= | {<process identifier> \| **this**} [24] |
| <stop> | ::= | **stop** |

## B.17 Decision

| | | |
|---|---|---|
| <decision> | ::= | **decision** <question> **;** <decision body> **enddecision** |
| <decision body> | ::= | { <answer part> <else part> } |
| | | \| { <answer part> {<answer part>}+ [<else part>] } |
| <answer part> | ::= | **(**<range condition>[25] **):**[<transition>] |
| <else part> | ::= | **else:**[<transition>] |
| <question> | ::= | **any** |
| | | \| <IExpr>$_{ta}$ |
| | | \| <IGuard>$_{ta}$ |
| <range condition> | ::= | <range> {**,** <range>}* |
| <range> | ::= | <IExpr>$_{ta}$ **:** <IExpr>$_{ta}$ |
| | | \| <IExpr>$_{ta}$ |
| | | \| <REL>$_{ta}$ <IExpr>$_{ta}$ |
| | | \| ! = <IExpr>$_{ta}$ |
| | | \| **true** |
| | | \| **false** |

**Semantic rules for decision. Rules:**

- A question must have least two answers.
- Answers must be mutually exclusive and should not overlap.
- One, and only one, answer may be an **else** answer, which is the complement of all other answers.
- The question and the answers or range conditions of the answers must either be of the same sort or informal text.
- If the question is of **any** type. Then all answers must be empty and no **else** answer can be used.

## B.18 Set/reset of timers

| | | |
|---|---|---|
| <set> | ::= | **set** <set statement> { **,**<set statement>} |
| <set statement> | ::= | **(** [ <IExpr>$_{ta}$**,**] <timer identifier> **)** |
| <reset> | ::= | **reset** <timer identifier> |

## B.19 Nextstate

| | | |
|---|---|---|
| <nextstate> | ::= | **nextstate** { <state identifier> \| <dash nextstate> } |
| <dash nextstate> | ::= | <hyphen> |
| <hyphen> | ::= | **–** |

## B.20 Join

| | | |
|---|---|---|
| <join> | ::= | **join** <connector name> |

## C Rules related to UPPAAL used in $\mathcal{SDL}_{xta}$

| | | |
|---|---|---|
| <IAssign List>$_{ta}$[26] | ::= | <IAssign>$_{ta}$ {,<IAssign>$_{ta}$ }* |
| <IAssign>$_{ta}$ | ::= | <ID>$_{ta}$ := <IExpr>$_{ta}$ |
| <IExpr>$_{ta}$ | ::= | <ID>$_{ta}$ |
| | | \| <ID>$_{ta}$ [<IExpr>$_{ta}$] |
| | | \| <NAT>$_{ta}$ |
| | | \| - <IExpr>$_{ta}$ |
| | | \| ( <IExpr>$_{ta}$ ) |
| | | \| <IExpr>$_{ta}$ <OP>$_{ta}$ <IExpr>$_{ta}$ |
| | | \| ( <IGuard>$_{ta}$ **?** <IExpr>$_{ta}$ **:** <IExpr>$_{ta}$ ) |
| <IGuard>$_{ta}$ | ::= | <IExpr>$_{ta}$ <REL>$_{ta}$ <IExpr>$_{ta}$ |
| | | \| <IExpr>$_{ta}$ **!=** <IExpr>$_{ta}$ |
| <REL>$_{ta}$ | ::= | **<** \| **<=** \| **>=** \| **>** \| **==** |
| <OP>$_{ta}$ | ::= | **+** \| **−** \| ***** \| **/** |
| <VIL>$_{ta}$ | ::= | <VID>$_{ta}$ { ,<VID>$_{ta}$ }* |
| <VID>$_{ta}$ | ::= | <ID>$_{ta}$ |
| | | \| <ID>$_{ta}$ **:=** <CExpr>$_{ta}$ |
| | | \| <ID>$_{ta}$ [<CExpr>$_{ta}$] |
| <CExpr>$_{ta}$ | ::= | <NAT>$_{ta}$ |
| | | \| <ID>$_{ta}$ |
| | | \| ( <CExpr>$_{ta}$ ) |
| | | \| <CExpr>$_{ta}$ <OP>$_{ta}$ <CExpr>$_{ta}$ |
| | | \| − <CExpr>$_{ta}$ |
| <param type>$_{ta}$ | ::= | **int** \| **int** [<NAT>$_{ta}$ **,** <NAT>$_{ta}$ ] |
| <LDecls>$_{ta}$ | ::= | { <LDecl>$_{ta}$ **;**}* |
| <LDecl>$_{ta}$ | ::= | { **int** <VIL>$_{ta}$ |
| | | \| **int** [<CExpr>$_{ta}$ **,** <CExpr>$_{ta}$ ] <VIL>$_{ta}$ |
| | | \| **const** <ID>$_{ta}$ <CExpr>$_{ta}$ } |

## D Usage of the conversion program

The conversion program can be downloaded from the World Wide Web at location

http://www.docs.uu.se/~hessel/sdl2xta/.

The invocation is:

sdl2xta <input-$\mathcal{SDL}_{xta}$-file> <output-.xta-file>

The output .xta file can be opened in UPPAAL. The user must unfortunately place the locations him-/herself because of the lack of graphical information in the .xta format.

## E The signal analysis algorithm

In this section we traverse the signal analysis algorithm step by step. We start from an output from one process set and find all possible receiver process sets. We have the triple <signal, via, to> that we bring with us all the time.

*Signal* is the name of the outputted signal that every signal route, channel or gate on our path must convey for our path to be valid. If the conveyed signal set is not specified then all signals can be conveyed. This is referred to as the *signal rule.*

*Via*, if it is specified is a list[27] of names that must match with gates, signal routes or channels in the given order. This is checked by removing names after they are seen. For a valid path all elements must have been removed. This is referred to as the *via rule.* If the first element match one of the possible gates, signal routes or channels in the scope, all other paths are invalid. We say that a path partially fulfills the *via rule*, if it is not invalid as described above.

*To*, if it is specified must be the name of the receiving process set. This is referred to as the *to rule.*

### E.1 Output in a process set

We start in the process set instance. If the process is typebased and have gates and if a via list is specified we check if the first via name is equal to one of the gates. If it is, then we remove the name for futher via processing and pass the gates name to the next step. The next step in the algorithm is to call *"Out from a process in a leaf block"*.

### E.2 Out from process in a leaf block

We are now in a leaf block and we follow an output from a process.

**No declared signal routes.** If there are no signal routes declared each of the other process sets is a receiver and is added to the result set if the *to* and the *via rule*s are fulfilled.

We also call the *"Out from block in a system or a substructure"* algorithm step for finding paths to process sets in other blocks. We add the result to the result set.

**There exist declared signal routes.** If there exist declared signal routes in the block then we want to find those signal routes that have an endpoint in the process set where we came from and the *signal rule* is fulfilled. If we have a non empty via list and one of the signal routes' name are equal to the first via element, then we follow only that signal route.

For each valid signal route we find we look at the other end point as when there didn't exist any signal routes. There are two different cases even here.

[27]Only one via name can be specified in $\mathcal{SDL}_{xta}$ at the time of writing.

**Endpoint goes to a process set.** The other endpoint goes to a process set, it is added to the result set if the *to* and the *via rule*s are fulfilled.

**Endpoint goes to the environment.** The other endpoint goes to the environment we call the *"Out from block in a system or a substructure"* algorithm step. If the block is typebased and the signal route is specified to go via a specific gate that gate is passed to the next step. If the block is not typebased and there is a connect for the signal route with a outer channel this channel is passed to the next step. Also the *via rule* must be partially fulfilled.

## E.3 Out from block in a system or a substructure

We are now in a system or in a substructure and are following a signal path from a known block, eventually we also know the name of the gate.

**No declared channels.** If channel declarations are omitted then we check all possibilities. The result set is what we add up from calling the following algorithm steps: *"Out form a substructure in a partitioned block"* (if this is not the system specification) and for the blocks found in this node *"Into a partitioned block"* or *"Into a leaf block"* depending on if the block is a partitial block or a leaf block.

**There exists channel declarations.** In case there are channel declarations valid channels must:

- have one endpoint at the block where we came from,
- be connected to the gate passed to this algorithm step (if a gate is passed),
- fulfill the *signal rule*,
- partially fulfill the *via rule*.

Now we look at the other endpoint. If the other endpoint is a block then we call one of the algorithm steps *"Into a partitioned block"* or *"Into a leaf block"* depending on if the block is a partitial block or a leaf block. If the channel we follow is connected to a gate, the gate is passed to the algorithm step. If the other endpoint goes upwards the hierarchy i.e. to the environment of the system or substructure, then the *"Out of substructure in partitional block"* algorithm step is called. If the channel we are following connects to a outer channel we must pass that value to the step.

## E.4 Out of substructure in partitional block

A partitioned block, has neither any process sets nor any signal routes, it has only a substructure. We call the *"Out from block in a system or a substructure"* algorithm step. If this step was called with a channel name we pass the name on.

## E.5 Into a partitioned block

A partitioned block has only a substructure. This step is only conveying the name of the outer channel that it possibly was called with to the *"Into a substructure"* algorithm step.

## E.6 Into a substructure

In this step we may have a outer channel name to find a connection for with an internal channel, if the substructure has channels. We also know that we have to go down the hierarchy because a channel is not allowed to go from the environment to the envirionment.

**No declared channels.** If channel declarations are omitted then the signal can possibly go to any of the blocks in the substructure. For each block we call, depending on if the block is a partitioned block or a leaf block, the *"Into a leaf block"* or the *"Into a partitioned block"* algorithm step.

**There exist channels.** If there exist channels, then valid channels must:

- have one endpoint of the channel is connected to the environment,
- fulfill the *signal rule*,
- partially fulfill the *via rule*,
- must be connected to the eventual outer channel passed to this step.

For each valid channel we call the block at the other endpoint. Depending on if the block is a partitioned block or a leaf block, the *"Into a leaf block"* or the *"Into a partitioned block"* algorithm step is taken. As parameters we pass the gate name if the channel is connected to the block via a gate and the name of the channel.

## E.7 Into a leaf block

When following a signal into a leaf block. We eventually have as parameters the connecting outer channel or the gate where the channel connected the block.

**No declared signal routes.** If no signal route is declared in the block, then all process sets that fulfill the *to* and the *via rule* are accepted as receivers.

**There exist signal routes** Process sets are accepted as receivers if they are the endpoint of a valid signal route. Valid signal routes must:

- be connected to the environment with one endpoint,
- be connected to the eventual outer channel, given as parameter to the algotithm step,
- be connected to the environment via an eventual block gate, given as parameter to the algotithm step,
- fulfill the *signal rule*,
- fulfill the *via rule*,
- fulfill the *to rule*,

## F The full $\mathcal{SDL}_{xta}$-code for the exchanger example

```
//
// Anders Hessel
// Sender to receiver example

system s;
  signal s1(int), ack1(int);

  block b1;
    process mysender referenced;
    signalroute sr1 from mysender to env with s1;
                    from env to mysender with ack1;
    connect ch1 and sr1;
  endblock;

  block b2;
    signal s2(int), ack2(int);
    process type exchanger referenced;
    process x : exchanger;
    process  myreceiver referenced;
    signalroute sr1 from x via g1 to myreceiver with s2;
                    from myreceiver to x via g1 with ack2;
    signalroute sr2 from x via g2 to env with ack1;
              from env to x via g2 with s1;
   connect ch1 and sr2;
  endblock;

  channel ch1 from b1 to b2 with s1;
              from b2 to b1 with ack1;
  endchannel;
endsystem;

process type exchanger;
 gate g1 out with s2; in with ack2;
 gate g2 in with s1; out with ack1;
 dcl int val1;
 dcl int val2;
 dcl int times := 0;
 pid sender1;

 start;
   nextstate wait1;
 state wait1;
   input s1(val1);
    task sender1 := sender [1,2];
    decision any;
    ():
       getsig:
       output s2(val1) via g1;
       nextstate wait2;
    ():
      // ooops lost signal
      task times := times + 1;
      decision times;
      (>3):
   join getsig;
      else:
        nextstate -;
      enddecision;
    enddecision;

 state wait2;
  input ack2(val2);
    // never loose ack
    output ack1(val2) to sender1;
    stop;
 endstate;
endprocess type;


process mysender;
 dcl const secretval 3;
 dcl int otherval;
 timer t1;

 start;
 label1 :
   output s1(secretval);
   set (10, t1);
   nextstate wait;

 state wait;
   input ack1(otherval);
     stop;
   input t1;
     join label1;
```

```
endprocess;


process myreceiver;
  dcl const mysecretval 1;
  dcl int secretval;
  dcl int[0,1] done := 0;

  start;
    nextstate wait;
  state wait;
    input s2(secretval);
      task done := 1 [1,5] ;
      output ack2(mysecretval);
      stop;
endprocess;
```

## G The full UPPAAL .xta-code for the exchanger example

Indentation is done after generation.

```
// UPPAAL xta-code generated by
// Anders Hessel's sdl2xta tool version 0.91a
// ISD Datasystem AB and Uppsala University, IT


const true 1;
const false 0;
const expl_sender 1;
urgent chan explsig, sdl_explicit, sdl_pid_remove;
int[0,4]  rem_pid, expl_to;
int[0,6] expl_signal;
int  expl_params[1];
int[0,1] qlive[4];
int[0,4] pids[21];
// Generating globals
const s1_1 1;
const ack1_2 2;
const s2_3 3;
const ack2_4 4;
urgent chan mysender_1_create;
urgent chan mysender_1_start;
urgent chan mysender_1_stop;
urgent chan mysender_1_psisig;
urgent chan mysender_1_implsig;
const mysender_1_ps_parent_index_2 2;
const mysender_1_pso_sender_index_3 3;
const mysender_1_psi_sender_index_4 4;
urgent chan mysender_1_0_t1_set;
urgent chan mysender_1_0_t1_reset;
int mysender_1_0_t1_set_time;
const mysender_1_0_t1_signal_val_5 5;
const mysender_self_1 1;
const mysender_parent_index_5 5;
const mysender_offspring_index_6 6;
const mysender_sender_index_7 7;
urgent chan mysender_1_0_nextsig;
urgent chan mysender_1_0_savesig;
urgent chan mysender_1_0_acceptsig;
urgent chan mysender_1_0_timersig;
urgent chan mysender_1_0_contsig;
urgent chan mysender_1_0_q_stop;
urgent chan mysender_1_0_q_start;
urgent chan x_2_create;
urgent chan x_2_start;
urgent chan x_2_stop;
urgent chan x_2_psisig;
urgent chan x_2_implsig;
const x_2_ps_parent_index_8 8;
const x_2_pso_sender_index_9 9;
const x_2_psi_sender_index_10 10;
urgent chan myreceiver_3_create;
urgent chan myreceiver_3_start;
urgent chan myreceiver_3_stop;
urgent chan myreceiver_3_psisig;
urgent chan myreceiver_3_implsig;
const myreceiver_3_ps_parent_index_11 11;
const myreceiver_3_pso_sender_index_12 12;
const myreceiver_3_psi_sender_index_13 13;
const sender1_14 14;
const x_self_2 2;
const x_parent_index_15 15;
const x_offspring_index_16 16;
const x_sender_index_17 17;
urgent chan x_2_0_nextsig;
urgent chan x_2_0_savesig;
urgent chan x_2_0_acceptsig;
urgent chan x_2_0_timersig;
urgent chan x_2_0_contsig;
urgent chan x_2_0_q_stop;
urgent chan x_2_0_q_start;
const myreceiver_self_3 3;
const myreceiver_parent_index_18 18;
const myreceiver_offspring_index_19 19;
const myreceiver_sender_index_20 20;
urgent chan myreceiver_3_0_nextsig;
urgent chan myreceiver_3_0_savesig;
urgent chan myreceiver_3_0_acceptsig;
urgent chan myreceiver_3_0_timersig;
urgent chan myreceiver_3_0_contsig;
urgent chan myreceiver_3_0_q_stop;
urgent chan myreceiver_3_0_q_start;
urgent chan mysender_1_output_s1;
int [0,21] mysender_1_ps_offspring;
int [0,6] mysender_1_pso_signal;
int [0,6] mysender_1_psi_signal;
int mysender_1_pso_params[1] ;
int mysender_1_psi_params[1] ;
int [0,6] mysender_1_0_in_signal;
int [0,6] mysender_1_0_tq_signal;
int [0,4] mysender_1_0_sender_queue[20] ;
int mysender_1_0_param[1] ;
urgent chan x_2_output_s2_via_g1;
int [0,21] x_2_ps_offspring;
int [0,6] x_2_pso_signal;
int [0,6] x_2_psi_signal;
int x_2_pso_params[1] ;
int x_2_psi_params[1] ;
int [0,6] x_2_0_in_signal;
int [0,6] x_2_0_tq_signal;
int [0,4] x_2_0_sender_queue[20] ;
int x_2_0_param[1] ;
urgent chan myreceiver_3_output_ack2;
int [0,21] myreceiver_3_ps_offspring;
int [0,6] myreceiver_3_pso_signal;
int [0,6] myreceiver_3_psi_signal;
int myreceiver_3_pso_params[1] ;
int myreceiver_3_psi_params[1] ;
int [0,6] myreceiver_3_0_in_signal;
int [0,6] myreceiver_3_0_tq_signal;
int [0,4] myreceiver_3_0_sender_queue[20] ;
int myreceiver_3_0_param[1] ;

process StartUp
{
 state myreceiver_3_create_0, mysender_1_create_0, Last, First,
  x_2_create_0;
 commit myreceiver_3_create_0, mysender_1_create_0, x_2_create_0;
 init First;

trans First -> mysender_1_create_0{
 sync mysender_1_create!;
},
mysender_1_create_0 -> x_2_create_0{
 sync x_2_create!;
},
x_2_create_0 -> myreceiver_3_create_0{
 sync myreceiver_3_create!;
},
myreceiver_3_create_0 -> Last{};
}

process Queue(
 const self;
 urgent chan nextsig, acceptsig, savesig, contsig;
 urgent chan implsig, timersig, start, stop;
 const p_sender, ps_sender;
 int[0,6] p_signal, ps_signal, tq_signal;
 int ps_params[1], p_params[1];
 int[0,4] sender[20])
{
 int[0,6] queue[20];
 int[0, 20] test;
 int[0, 20] next;
 int[0, 20] mover;
 int params[20];
 int[0,1] cont;
 state NewTimer, New, Wait, Inactive, Wait_accept, NewImpl, Move, NewExpl;
 commit NewTimer, New, NewImpl, NewExpl;
 urgent Wait_accept, Move;
 init Inactive;

trans Wait -> Wait_accept{
 guard next != test; sync nextsig?;
 assign p_signal := queue[test],
  pids[p_sender]:=sender[test],
  p_params[0] := params[0 + test *1];
},
Wait_accept -> Move{
 sync acceptsig?;
 assign mover:=test;
},
Wait -> NewImpl{
 sync implsig?;
 assign sender[next] := pids[ps_sender],
  params[next+0] := ps_params[0],
  queue[next] := ps_signal;
},
NewImpl -> New{
 assign pids[ps_sender] := 0,
  ps_params[0] := 0,
  ps_signal := 0;
},
Wait -> NewExpl{
 guard self == expl_to;
 sync explsig?;
 assign sender[next] := pids[expl_sender],
 params[next+0] := expl_params[0],
 queue[next] := expl_signal;
},
NewExpl -> New{
 assign pids[expl_sender] := 0,
  expl_params[0] := 0,
  expl_signal := 0,
  expl_to := 0;
},
Wait -> NewTimer{
 sync timersig?;
 assign sender[next] := self,
  queue[next] := tq_signal;
},
NewTimer -> New{
 assign params[next+0] := 0,
  tq_signal := 0;
},
New -> Wait{
 assign next:=next+1,
  cont:=true;
},
Inactive -> Wait{
 sync start?;
 assign qlive[self] :=true;
},
Wait -> Inactive{
 sync stop?;
 assign qlive[self] :=false;
},
Move -> Move{
 guard mover+1 != next;
 assign sender[mover] := sender [mover+1],
 queue[mover] := queue[mover +1],
 params[mover+0] := params[mover+1+0],
 mover := mover +1;
},
Wait_accept -> Wait{
 sync savesig?;
 assign test:=test+1;
},
Move -> Wait{
 guard mover +1 == next;
 assign sender[next-1] := 0,
  queue[next-1]:=0,
  params[next-1+0] := 0,
  next:= next -1,
  test := 0;
},
Wait -> Wait{
 guard test == next, cont == true;
 sync contsig?;
 assign cont := false,
 pids[p_sender]:=self;
};
}

process Timer(
 urgent chan set, reset;
 urgent chan timersig;
 int set_time;
 const timer;
 int[0,6] tq_signal)
{
 clock c;
 int time;
 state Unset, Set{c <= 1}, Test;
 urgent Test;
 init Unset;

trans Unset -> Test{
 sync set?;
 assign time:=0;
},
Set -> Test{
 guard c == 1;
 assign time := time+1;
},
Test -> Unset{
 guard time == set_time;
 sync timersig!;
 assign tq_signal:=timer;
},
Test -> Set{
 guard time<set_time;
 assign c:= 0;
},
Set -> Unset{
 sync reset?;
},
Set -> Test{
 sync set?;
 assign time:=0;
};
}

process Expl{
 state Have_signal, Wait;
 urgent Have_signal;
 init Wait;

trans Wait -> Have_signal{
 sync sdl_explicit?;
},
Have_signal -> Wait{
 guard qlive[expl_to] ==false;
 assign expl_to :=0,
  expl_signal :=0,
  expl_params[0] := 0;
},
Have_signal -> Wait{
 sync explsig!;
};
}

process RemPid
{
 int count := 0;
 state RemPids_end, x_2_0_sender_queue_end, RemPids, x_2_0_sender_queue_do,
  mysender_1_0_sender_queue_do, myreceiver_3_0_sender_queue_end, Wait,
  myreceiver_3_0_sender_queue_do, mysender_1_0_sender_queue_end;
 commit RemPids_end, x_2_0_sender_queue_end, RemPids, x_2_0_sender_queue_do,
  mysender_1_0_sender_queue_do, myreceiver_3_0_sender_queue_end,
  myreceiver_3_0_sender_queue_do, mysender_1_0_sender_queue_end;
 init Wait;

trans Wait -> RemPids{
 sync sdl_pid_remove?;
 assign count := 0;
},
RemPids -> RemPids{
 guard count < 21,
  pids[count] != rem_pid;
 assign count := count + 1;
},
RemPids -> RemPids{
 guard count < 21,
  pids[count] == rem_pid;
 assign pids[count] := 0,
  count := count + 1;
},
RemPids -> RemPids_end{
 guard count == 21;
 assign count := 0;
},
RemPids_end -> mysender_1_0_sender_queue_do{},
mysender_1_0_sender_queue_do -> mysender_1_0_sender_queue_do{
 guard count < 20,
  mysender_1_0_sender_queue[count] != rem_pid;
 assign count := count + 1;
},
mysender_1_0_sender_queue_do -> mysender_1_0_sender_queue_do{
 guard count < 20,
  mysender_1_0_sender_queue[count] == rem_pid;
 assign mysender_1_0_sender_queue[count] :=0,
  count := count + 1;
},
mysender_1_0_sender_queue_do -> mysender_1_0_sender_queue_end{
 guard count == 20;
 assign count := 0;
},
mysender_1_0_sender_queue_end -> x_2_0_sender_queue_do{},
x_2_0_sender_queue_do -> x_2_0_sender_queue_do{
 guard count < 20,
  x_2_0_sender_queue[count] != rem_pid;
 assign count := count + 1;
},
x_2_0_sender_queue_do -> x_2_0_sender_queue_do{
 guard count < 20,
  x_2_0_sender_queue[count] == rem_pid;
 assign x_2_0_sender_queue[count] :=0,
  count := count + 1;
},
x_2_0_sender_queue_do -> x_2_0_sender_queue_end{
 guard count == 20;
 assign count := 0;
},
x_2_0_sender_queue_end -> myreceiver_3_0_sender_queue_do{},
myreceiver_3_0_sender_queue_do -> myreceiver_3_0_sender_queue_do{
 guard count < 20,
  myreceiver_3_0_sender_queue[count] != rem_pid;
 assign count := count + 1;
},
myreceiver_3_0_sender_queue_do -> myreceiver_3_0_sender_queue_do{
 guard count < 20,
  myreceiver_3_0_sender_queue[count] == rem_pid;
 assign myreceiver_3_0_sender_queue[count] :=0,
  count := count + 1;
},
myreceiver_3_0_sender_queue_do -> myreceiver_3_0_sender_queue_end{
 guard count == 20;
 assign count := 0;
},
myreceiver_3_0_sender_queue_end -> Wait{
 guard rem_pid == expl_to;
 assign expl_to :=0;
},
myreceiver_3_0_sender_queue_end -> Wait{
 guard rem_pid != expl_to;
};
}

process PSI(
 const maxval;
 urgent chan sdl_create, sdl_start, sdl_stop;
 urgent chan psisig, implsig;
 int[0,21] offspring)
{
 int[0,1] started :=0;
 state Create, Have_signal, Wait;
 commit Create;
 urgent Have_signal;
 init Wait;

trans Wait -> Wait{
 sync sdl_stop?;
 assign started := started - 1;
},
Wait -> Wait{
 guard started == maxval;
 sync sdl_create?;
 assign pids[offspring] := 0;
},
Wait -> Create{
 guard started < maxval;
 sync sdl_create?;
},
Create -> Wait{
 sync sdl_start!;
 assign started := started +1;
},
Wait -> Have_signal{
 guard started > 0;
 sync psisig?; },
Have_signal -> Wait{
 sync implsig!;
};
}

process mysender(
 const self;
 const parent, offspring, sender;
 const pso_sender;
 urgent chan sdl_start, sdl_stop;
 int[0,21] ps_offspring;
 const  ps_parent;
 // PidParam
 // no local pids
 // ExternalParam (non local signals)
 const ack1, s1;
 const t1;
 // TimerParam (non local signals)
 urgent chan t1_set, t1_reset;
 int t1_set_time;
 // OutSyncParam (output sync params)
 urgent chan output_s1;
 // CreateSyncParam (create sync params)
 // PI -> Queue and Timer -> Queue
 urgent chan nextsig, acceptsig, savesig, contsig, q_start, q_stop;
 // param holders in and out (out is for the PS)
 int in_params [1], pso_params [1];
 // signal holders in and out (out is for the PS)
 int[0,6] in_signal, pso_signal)
{
 clock c;
 const secretval 3;
 int otherval;
 const wait_Val 0;
 int[0,1] iSDL_State;
 state iSDL_Start, iSDL_Inactive, join2, iSDL_State_choser, iSDL_RemPid,
  label1, SDL_wait, set0, iSDL_Reset, SDL_Save_test_wait, iSDL_Stop,
  output2, iSDL_StopQueue, stop1, nextstate4;
 urgent iSDL_Start, iSDL_Inactive, join2, iSDL_State_choser, iSDL_RemPid,
  label1, set0, iSDL_Reset, iSDL_Stop, output2, iSDL_StopQueue, stop1,
  nextstate4;
 init iSDL_Inactive;

trans iSDL_Inactive -> iSDL_Start{
 sync sdl_start?;
 assign t1_set_time:= 0,
 pids[ps_offspring] := self,
 pids[parent] := pids[ps_parent];
},
iSDL_State_choser -> SDL_wait{
 guard iSDL_State == wait_Val;
},
SDL_wait -> SDL_Save_test_wait{
 sync nextsig!;
},
SDL_Save_test_wait -> stop1{
 guard in_signal == ack1;
 sync acceptsig!;
 assign otherval := in_params[0];
},
SDL_Save_test_wait -> join2{
 guard in_signal == t1;
 sync acceptsig!;
},
SDL_Save_test_wait -> SDL_wait{
 guard in_signal != ack1,
 in_signal != t1; sync acceptsig!;
},
label1 -> output2{},
output2 -> set0{
 sync output_s1!;
 assign pso_signal:=s1,
  pso_params[0] := secretval,
  pids[pso_sender] := self;
},
set0 -> nextstate4{
 sync t1_set!;
 assign t1_set_time:=10;
},
nextstate4 -> iSDL_State_choser{
 assign iSDL_State :=wait_Val;
},
stop1 -> iSDL_Stop{},
join2 -> label1{},
iSDL_Start -> label1{
 sync q_start!;
},
iSDL_StopQueue -> iSDL_RemPid{
 sync q_stop!;
},
iSDL_RemPid -> iSDL_Reset{
 sync sdl_pid_remove!;
 assign rem_pid := self;
},
iSDL_Reset -> iSDL_Inactive{
 assign otherval:= 0,
  t1_set_time:= 0,
  pids[offspring] :=0,
  pids[sender] := 0;
};
}

process exchanger(
 const self;
 const parent, offspring, sender;
 const pso_sender;
 urgent chan sdl_start, sdl_stop;
 int[0,21] ps_offspring;
 const  ps_parent;
 // PidParam
 const sender1;
 // ExternalParam (non local signals)
 const ack1, ack2, s1, s2;
 // TimerParam (non local signals)
 // OutSyncParam (output sync params)
 urgent chan output_s2_via_g1;
 // CreateSyncParam (create sync params)
 // PI -> Queue and Timer -> Queue
 urgent chan nextsig, acceptsig, savesig, contsig, q_start, q_stop;
 // param holders in and out (out is for the PS)
 int in_params [1], pso_params [1];
 // signal holders in and out (out is for the PS)
 int[0,6] in_signal, pso_signal)
{
 clock c;
 int val1;
 int val2;
 int times:= 0;
 const wait1_Val 0;
 const wait2_Val 1;
 int[0,2] iSDL_State;
 state task1, task0{c <=2}, nextstate3, nextstate2, iSDL_State_choser,
  nextstate1, nextstate0, getsig, decision1, decision0, iSDL_StopQueue,
  SDL_Save_test_wait2, join1, join0, SDL_Save_test_wait1, iSDL_RemPid,
  output1, output0, iSDL_Reset, stop0, iSDL_Stop, enddecision0,
  iSDL_Inactive, SDL_wait2, SDL_wait1, iSDL_Start;
 urgent task1, nextstate3, nextstate2, iSDL_State_choser, nextstate1,
  nextstate0, getsig, decision1, decision0, iSDL_StopQueue, join1, join0,
  iSDL_RemPid, output1, output0, iSDL_Reset, stop0, iSDL_Stop,
  enddecision0, iSDL_Inactive, iSDL_Start;
 init iSDL_Inactive;

trans iSDL_Inactive -> iSDL_Start{
 sync sdl_start?;
 assign pids[ps_offspring] := self,
  pids[parent] := pids[ps_parent];
},
iSDL_State_choser -> SDL_wait1{
 guard iSDL_State == wait1_Val;
},
SDL_wait1 -> SDL_Save_test_wait1{
 sync nextsig!;
},
SDL_Save_test_wait1 -> task0{
 guard in_signal == s1;
 sync acceptsig!;
 assign c:=0,
  val1 := in_params[0];
},
SDL_Save_test_wait1 -> SDL_wait1{
 guard in_signal != s1;
 sync acceptsig!;
},
iSDL_State_choser -> SDL_wait2{
 guard iSDL_State == wait2_Val;
},
SDL_wait2 -> SDL_Save_test_wait2{
 sync nextsig!;
},
SDL_Save_test_wait2 -> output1{
 guard in_signal == ack2;
 sync acceptsig!;
 assign val2 := in_params[0];
},
SDL_Save_test_wait2 -> SDL_wait2{
 guard in_signal != ack2;
 sync acceptsig!;
},
nextstate0 -> iSDL_State_choser{
 assign iSDL_State :=wait1_Val;
},
task0 -> decision0{
 guard c >=1;
 assign pids[sender1]:=pids[sender];
},
decision0 -> getsig{},
getsig -> output0{},
output0 -> nextstate1{
 sync output_s2_via_g1!;
 assign pso_signal:=s2,
  pso_params[0] := val1,
  pids[pso_sender] := self;
},
nextstate1 -> iSDL_State_choser{
 assign iSDL_State :=wait2_Val;
},
decision0 -> task1{},
task1 -> decision1{
 assign times:=times+1;
},
decision1 -> join0{
 guard (times>3? 1 : 0) != 0;
},
join0 -> getsig{},
decision1 -> nextstate2{
 guard ((times>3? 1 : 0)) == 0;
},
nextstate2 -> iSDL_State_choser{},
join1 -> enddecision0{},
enddecision0 -> nextstate3{},
nextstate3 -> iSDL_State_choser{},
output1 -> stop0{
 sync sdl_explicit!;
 assign expl_signal:=ack1,
  expl_to := pids[sender1],
  expl_params[0] := val2,
  pids[expl_sender] := self;
},
stop0 -> iSDL_Stop{},
iSDL_Start -> nextstate0{
 sync q_start!;
},
iSDL_Stop -> iSDL_StopQueue{
 sync sdl_stop!;
},
iSDL_StopQueue -> iSDL_RemPid{
 sync q_stop!;
},
iSDL_RemPid -> iSDL_Reset{
 sync sdl_pid_remove!;
 assign rem_pid := self;
},
iSDL_Reset -> iSDL_Inactive{
 assign val1:= 0,
  val2:= 0,
  times:= 0,
  pids[offspring] :=0,
  pids[sender] := 0;
};
}

process myreceiver(
 const self;
 const parent, offspring, sender;
 const pso_sender;
 urgent chan sdl_start, sdl_stop;
 int[0,21] ps_offspring;
 const  ps_parent;
 // PidParam
 // no local pids
 // ExternalParam (non local signals)
 const ack2, s2;
 // TimerParam (non local signals)
 // OutSyncParam (output sync params)
 urgent chan output_ack2;
 // CreateSyncParam (create sync params)
 // PI -> Queue and Timer -> Queue
 urgent chan nextsig, acceptsig, savesig, contsig, q_start, q_stop;
 // param holders in and out (out is for the PS)
 int in_params [1], pso_params [1];
 // signal holders in and out (out is for the PS)
 int[0,6] in_signal, pso_signal)
{
 clock c;
 const mysecretval 1;
 int secretval;
 int [0,1] done:= 0;
 const wait_Val 0;
 int[0,1] iSDL_State;
 state iSDL_Start, iSDL_Inactive, task2{c <=5}, iSDL_State_choser,
  iSDL_RemPid, SDL_wait, iSDL_Reset, SDL_Save_test_wait, iSDL_Stop,
  output3, stop2, iSDL_StopQueue, nextstate5;
 urgent iSDL_Start, iSDL_Inactive, iSDL_State_choser, iSDL_RemPid,
  iSDL_Reset, iSDL_Stop, output3, stop2, iSDL_StopQueue, nextstate5;
 init iSDL_Inactive;

trans iSDL_Inactive -> iSDL_Start{
 sync sdl_start?;
 assign pids[ps_offspring] := self,
  pids[parent] := pids[ps_parent];
},
iSDL_State_choser -> SDL_wait{
 guard iSDL_State == wait_Val;
},
SDL_wait -> SDL_Save_test_wait{
 sync nextsig!;
},
SDL_Save_test_wait -> task2{
 guard in_signal == s2;
 sync acceptsig!;
 assign c:=0,
  secretval := in_params[0];
},
SDL_Save_test_wait -> SDL_wait{
 guard in_signal != s2;
 sync acceptsig!;
},
nextstate5 -> iSDL_State_choser{
 assign iSDL_State :=wait_Val;
},
task2 -> output3{
 guard c >=1;
 assign done:=1;
},
output3 -> stop2{
 sync output_ack2!;
 assign pso_signal:=ack2,
  pso_params[0] := mysecretval,
  pids[pso_sender] := self;
},
stop2 -> iSDL_Stop{},
iSDL_Start -> nextstate5{
 sync q_start!;
},
iSDL_Stop -> iSDL_StopQueue{
 sync sdl_stop!;
```

```
},
iSDL_StopQueue -> iSDL_RemPid{
 sync q_stop!;
},
iSDL_RemPid -> iSDL_Reset{
 sync sdl_pid_remove!;
 assign rem_pid := self;
},
iSDL_Reset -> iSDL_Inactive{
 assign secretval:= 0,
  done:= 0,
  pids[offspring] :=0,
  pids[sender] := 0;
};
}

process mysender_1_PS(
 const pso_sender;
 int[0,6] pso_signal;
 int pso_params[1];
 urgent chan output_s1;
 urgent chan x_2_psisig;
 const x_2_sender;
 int x_2_params[1];
 int[0,6] x_2_signal)
{
 state Got_output_s1, Wait, Clean;
 commit Clean;
 urgent Got_output_s1;
 init Wait;

trans Wait -> Got_output_s1{
 sync output_s1?;
},
Got_output_s1 -> Clean{
 sync x_2_psisig!;
 assign x_2_signal := pso_signal,
 x_2_params[0] := pso_params[0],
 pids[x_2_sender] := pids[pso_sender];
},
Got_output_s1 -> Clean{
 guard qlive[x_self_2] == false;
},
Clean -> Wait{
 assign pso_signal := 0,
  pso_params[0] := 0,
 pids[pso_sender] := 0;
};
}

process x_2_PS(
 const pso_sender;
 int[0,6] pso_signal;
 int pso_params[1];
 urgent chan output_s2_via_g1;
 urgent chan myreceiver_3_psisig;
 const myreceiver_3_sender;
 int myreceiver_3_params[1];
 int[0,6] myreceiver_3_signal)
{
 state Wait, Got_output_s2_via_g1, Clean;
 commit Clean;
 urgent Got_output_s2_via_g1;
 init Wait;

trans Wait -> Got_output_s2_via_g1{
 sync output_s2_via_g1?;
},
Got_output_s2_via_g1 -> Clean{
 sync myreceiver_3_psisig!;
 assign myreceiver_3_signal := pso_signal,
  myreceiver_3_params[0] := pso_params[0],
  pids[myreceiver_3_sender] := pids[pso_sender];
},
Got_output_s2_via_g1 -> Clean{
 guard qlive[myreceiver_self_3] == false;
},
Clean -> Wait{
 assign pso_signal := 0,
  pso_params[0] := 0,
  pids[pso_sender] := 0;
};
}

process myreceiver_3_PS(
 const pso_sender;
 int[0,6] pso_signal;
 int pso_params[1];
 urgent chan output_ack2;
 urgent chan x_2_psisig;
 const x_2_sender;
 int x_2_params[1];
 int[0,6] x_2_signal)
{
state Wait, Got_output_ack2, Clean;
commit Clean;
urgent Got_output_ack2;
init Wait;

trans Wait -> Got_output_ack2{
 sync output_ack2?;
},
Got_output_ack2 -> Clean{
 sync x_2_psisig!;
 assign x_2_signal := pso_signal,
  x_2_params[0] := pso_params[0],
  pids[x_2_sender] := pids[pso_sender];
},
Got_output_ack2 -> Clean{
 guard qlive[x_self_2] == false;
},
Clean -> Wait{
 assign pso_signal := 0,
  pso_params[0] := 0,
  pids[pso_sender] := 0;
};
}

mysender_1_PSI := PSI( 1, mysender_1_create, mysender_1_start,
 mysender_1_stop, mysender_1_psisig, mysender_1_implsig,
 mysender_1_ps_offspring);

mysender_1_PSO:=mysender_1_PS(mysender_1_pso_sender_index_3,
 mysender_1_pso_signal,
 mysender_1_pso_params,
 mysender_1_output_s1, x_2_psisig, x_2_psi_sender_index_10,
 x_2_psi_params, x_2_psi_signal);

// GENERATE mysender_1_0
mysender_1_0 := mysender(mysender_self_1, mysender_parent_index_5,
 mysender_offspring_index_6, mysender_sender_index_7,
 mysender_1_pso_sender_index_3, mysender_1_start, mysender_1_stop,
 mysender_1_ps_offspring, mysender_1_ps_parent_index_2, ack1_2, s1_1,
 mysender_1_0_t1_signal_val_5, mysender_1_0_t1_set, mysender_1_0_t1_reset,
 mysender_1_0_t1_set_time, mysender_1_output_s1, mysender_1_0_nextsig,
 mysender_1_0_acceptsig, mysender_1_0_savesig, mysender_1_0_contsig,
 mysender_1_0_q_start, mysender_1_0_q_stop, mysender_1_0_param,
 mysender_1_pso_params,
 mysender_1_0_in_signal, mysender_1_pso_signal);

// GENERATE mysender_1_0 QUEUE
mysender_1_0_queue := Queue(mysender_self_1, mysender_1_0_nextsig,
 mysender_1_0_acceptsig, mysender_1_0_savesig, mysender_1_0_contsig,
 mysender_1_implsig, mysender_1_0_timersig, mysender_1_0_q_start,
 mysender_1_0_q_stop, mysender_sender_index_7,
 mysender_1_psi_sender_index_4, mysender_1_0_in_signal,
 mysender_1_psi_signal, mysender_1_0_tq_signal, mysender_1_psi_params,
 mysender_1_0_param, mysender_1_0_sender_queue);

// GENERATE mysender_1_0 TIMERS
mysender_1_0_t1 := Timer(mysender_1_0_t1_set, mysender_1_0_t1_reset,
 mysender_1_0_timersig, mysender_1_0_t1_set_time,
 mysender_1_0_t1_signal_val_5, mysender_1_0_tq_signal);
// END GENERATE mysender_1_0

x_2_PSI := PSI( 1, x_2_create, x_2_start, x_2_stop,
 x_2_psisig, x_2_implsig, x_2_ps_offspring);
 x_2_PSO:=x_2_PS(x_2_pso_sender_index_9,
 x_2_pso_signal,
 x_2_pso_params,
 x_2_output_s2_via_g1, myreceiver_3_psisig,
 myreceiver_3_psi_sender_index_13, myreceiver_3_psi_params,
 myreceiver_3_psi_signal);

// GENERATE x_2_0
x_2_0 := exchanger(x_self_2, x_parent_index_15, x_offspring_index_16,
 x_sender_index_17, x_2_pso_sender_index_9, x_2_start, x_2_stop,
 x_2_ps_offspring, x_2_ps_parent_index_8, sender1_14, ack1_2,
 ack2_4, s1_1, s2_3, x_2_output_s2_via_g1, x_2_0_nextsig,
 x_2_0_acceptsig, x_2_0_savesig, x_2_0_contsig, x_2_0_q_start,
 x_2_0_q_stop, x_2_0_param, x_2_pso_params,
 x_2_0_in_signal, x_2_pso_signal);

// GENERATE x_2_0 QUEUE
x_2_0_queue := Queue(x_self_2, x_2_0_nextsig, x_2_0_acceptsig,
 x_2_0_savesig, x_2_0_contsig, x_2_implsig, x_2_0_timersig,
 x_2_0_q_start, x_2_0_q_stop, x_sender_index_17, x_2_psi_sender_index_10,
 x_2_0_in_signal, x_2_psi_signal, x_2_0_tq_signal, x_2_psi_params,
 x_2_0_param, x_2_0_sender_queue);

// GENERATE x_2_0 TIMERS
// END GENERATE x_2_0

myreceiver_3_PSI := PSI( 1, myreceiver_3_create, myreceiver_3_start,
 myreceiver_3_stop, myreceiver_3_psisig, myreceiver_3_implsig,
 myreceiver_3_ps_offspring);
 myreceiver_3_PSO:=myreceiver_3_PS(myreceiver_3_pso_sender_index_12,
 myreceiver_3_pso_signal,
 myreceiver_3_pso_params,
 myreceiver_3_output_ack2, x_2_psisig, x_2_psi_sender_index_10,
 x_2_psi_params, x_2_psi_signal);

// GENERATE myreceiver_3_0
myreceiver_3_0 := myreceiver(myreceiver_self_3, myreceiver_parent_index_18,
 myreceiver_offspring_index_19, myreceiver_sender_index_20,
 myreceiver_3_pso_sender_index_12, myreceiver_3_start, myreceiver_3_stop,
 myreceiver_3_ps_offspring, myreceiver_3_ps_parent_index_11, ack2_4, s2_3,
  myreceiver_3_output_ack2, myreceiver_3_0_nextsig,
 myreceiver_3_0_acceptsig, myreceiver_3_0_savesig, myreceiver_3_0_contsig,
 myreceiver_3_0_q_start, myreceiver_3_0_q_stop, myreceiver_3_0_param,
 myreceiver_3_pso_params,
 myreceiver_3_0_in_signal, myreceiver_3_pso_signal);

// GENERATE myreceiver_3_0 QUEUE
myreceiver_3_0_queue := Queue(myreceiver_self_3, myreceiver_3_0_nextsig,
 myreceiver_3_0_acceptsig, myreceiver_3_0_savesig, myreceiver_3_0_contsig,
 myreceiver_3_implsig, myreceiver_3_0_timersig, myreceiver_3_0_q_start,
 myreceiver_3_0_q_stop, myreceiver_sender_index_20,
 myreceiver_3_psi_sender_index_13, myreceiver_3_0_in_signal,
 myreceiver_3_psi_signal, myreceiver_3_0_tq_signal,
 myreceiver_3_psi_params, myreceiver_3_0_param,
 myreceiver_3_0_sender_queue);

// GENERATE myreceiver_3_0 TIMERS
// END GENERATE myreceiver_3_0
expl := Expl();
startUp := StartUp();
remPid := RemPid();

// SYSTEM
```

```
system mysender_1_PSI, mysender_1_0, mysender_1_0_queue, mysender_1_0_t1,
 x_2_PSI, myreceiver_3_PSI, x_2_0, x_2_0_queue, myreceiver_3_0,
 myreceiver_3_0_queue, mysender_1_PSO, x_2_PSO, myreceiver_3_PSO, expl,
 startUp, remPid;
```