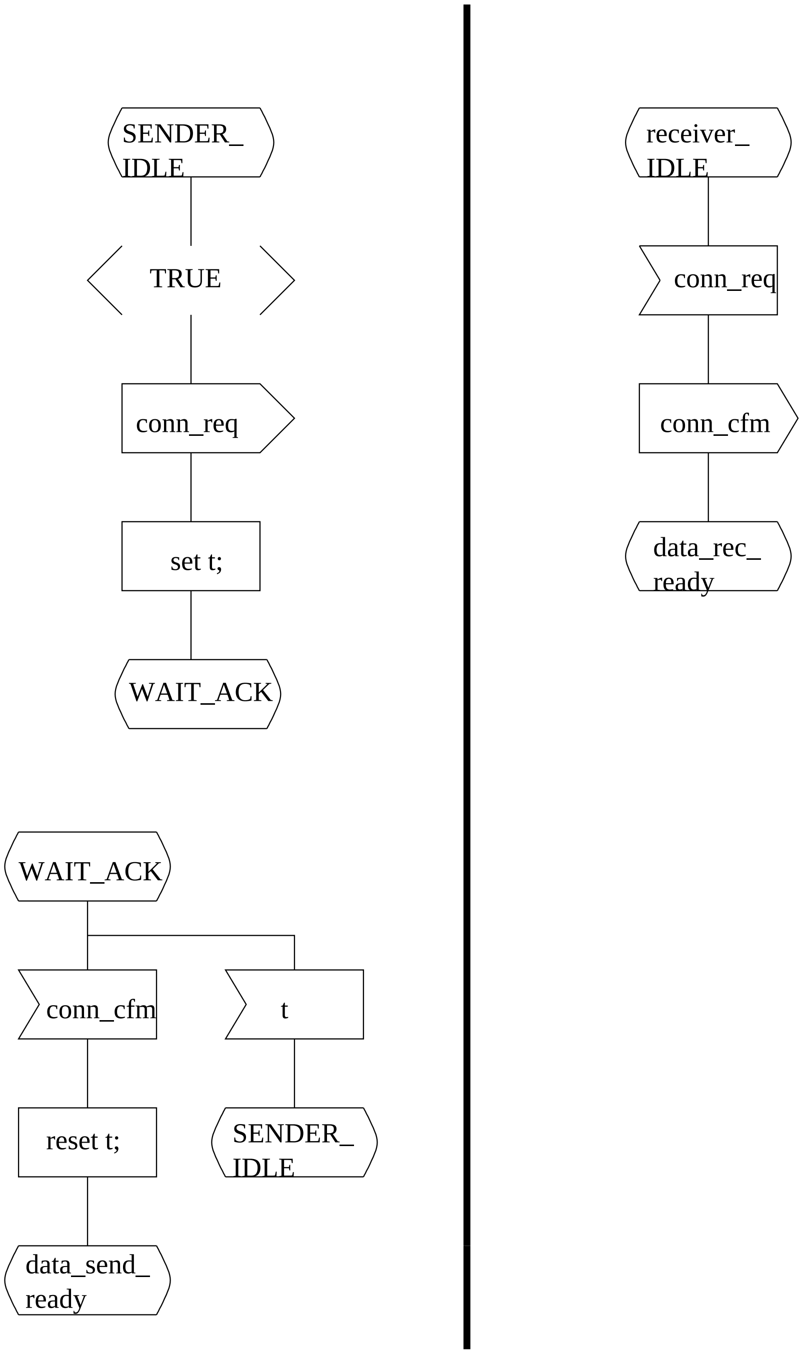
Tutorial #7
T-79.231 Parallel and distributed digital systems Nov 13, 2001
Teemu Tynjälä, Jukka Honkola

How to convert an SDL specification to Maria language



A simple communication protocol

Translating SDL specifications to Maria input language is fairly straightforward. SDL data types are easy to translate into Maria types. There is also a queue type in Maria, which will be used to represent the input queue of the process. In general, it is possible to translate one SDL statement to one Maria transition.

First we create the places:

- Create a control place. This place will hold information about the states of the processes. It will have tokens of the type `{pid,state}` where `pid` is the process id and `state` is the state that the process is in.
- Create places for variables. There will be one place for each variable of a process type. The specification in this tutorial does not have variables.
- Create a place for queues. This place will hold all the queues of the processes. The queues are differentiated by pids.
- Create a place for timers. The tokens will be of form `{pid,active?}` where `active?` is a boolean variable which is true if the timer is active.

After places, we convert the SDL statements into Maria transitions:

All: Everything we do will advance the control of the process in question.

Output: Put message into appropriate queue, check that there is room in the queue. This can be done with gate expression `gate %q > 0`, where `%` is the operator which gives out remaining free slots in the queue

Input: Process can read from its own queue. Often the subsequent actions depend on the message that was read from the queue. By using gate-expression, it is possible to use different transitions for different messages. The transitions should be mutually exclusive, that is, only one transition for each message. Also, one might want to check that there are elements in the queue.

Timers: Handling time in Petri Nets is difficult, as the nets do not have any concept of time. The timers are in this case modelled by considering both alternatives (the timer signal is received by the process or the timer is reset) at the point where the timer signal would be read from the input queue. This approach works well in many cases.

Variables: Straightforward because Maria supports most of the types in SDL directly.

In addition to these general guidelines, one might want to make some additional modifications to the net. For example, in this tutorial the processes are assumed to use a lossy communications channel, which is modelled accordingly. This increases the number of transitions used to model the sending of message to two, one where the message gets to the destination and one where the message is lost along the way.

First, define the necessary types and places for the system.

`pid_t` is for process ids. Every process has to have an unique identifier. There are two processes, so the type will have two possible values (0 and 1).

`pc_t` enumerates all the states of the processes.

`control_t` is the type for the `control` place

`token_t` is the type for black token

`protocol_pdus` is the type used for the input queues of the processes. It is modeled as an union type with two possible fields, one for each message

`protocol_and_queue_type` is the type for the actual input queues. It is a struct with the `pid` and the queue. It is the type for the `lossy_channel`.

`time_t` is the type for timers. It consists of `pid` and a boolean variable which is set to true if the timer is active.

```
typedef unsigned (0..1) pid_t;

typedef enum{
sender_idle_0,
sender_idle_2,
sender_idle_3,
sender_wait_ack_0,
sender_wait_ack_2,
sender_data_send_ready_0,
receiver_idle_0,
receiver_idle_2,
receiver_data_rec_ready_0
} pc_t;

typedef struct { pid_t pid; pc_t pc; } control_t;

place control control_t : {0, sender_idle_0}, {1, receiver_idle_0};

typedef struct {} token_t;

typedef union {
     token_t conn_req;
     token_t conn_cfm;
} protocol_pdus;
```

```
typedef protocol_pdus[queue 1] protocol_pdus_channel;

typedef struct {pid_t pid; protocol_pdus_channel ch;} protocol_and_queue_type;

place lossy_channel protocol_and_queue_type :
{0, is protocol_pdus_channel {}}, {1, is protocol_pdus_channel {}};

typedef struct { pid_t pid; bool on_or_off; } time_t;

place Timer_t time_t : {0, false}, {1, false};
```

Here the sender side transitions are defined. Note that the output statement is modelled with two transitions: one where the message is transferred correctly and one where the message is lost.

```
trans sender_idle_1
in {
 control: {pid, sender_idle_0};
}
out {
 control: {pid, sender_idle_2};
};

trans sender_idle_2_message_transferred_correctly
in {
 control: {pid, sender_idle_2};
 lossy_channel: {+pid, q};
}
out {
 control: {pid, sender_idle_3};
 lossy_channel: {+pid, q + is protocol_pdus conn_req = {}};
} gate (%q > 0);

trans sender_idle_2_message_lost
in {
 control: {pid, sender_idle_2};
}
out {
 control: {pid, sender_idle_3};
};
```

```
trans sender_idle_3
in {
 control: {pid, sender_idle_3};
 Timer_t: {pid, false};
}
out {
 control: {pid, sender_wait_ack_0};
 Timer_t: {pid, true};
};

trans sender_wait_ack_1
in {
 control: {pid, sender_wait_ack_0};
 lossy_channel: {pid, q};
}
out {
 control: {pid, sender_wait_ack_2};
 lossy_channel: {pid, -q};
} gate (/q > 0 && *q is conn_cfm);

trans sender_wait_ack_2
in {
 control: {pid, sender_wait_ack_2};
 Timer_t: {pid, true};
}
out {
 control: {pid, sender_data_send_ready_0};
 Timer_t: {pid, false};
};

trans sender_wait_ack_3
in {
 control: {pid, sender_wait_ack_0};
 Timer_t: {pid, true};
}
out {
 control: {pid, sender_idle_0};
 Timer_t: {pid, false};
};

trans sender_data_send_ready_1
in {
 control: {pid, sender_data_send_ready_0};
 lossy_channel: {pid, q};
}
```

```
out {
 control: {pid, sender_data_send_ready_0};
 lossy_channel: {pid, -q};
} gate (/q > 0);
```

Here are the receiver side transitions.

```
trans receiver_idle_1
in {
 control: {pid, receiver_idle_0};
 lossy_channel: {pid, q};
}
out {
 control: {pid, receiver_idle_2};
 lossy_channel: {pid, -q};
} gate (/q > 0 && *q is conn_req);

trans receiver_idle_2_message_not_lost
in {
 control: {pid, receiver_idle_2};
 lossy_channel: {+pid, q};
}
out {
 control: {pid, receiver_data_rec_ready_0};
 lossy_channel: {+pid, q + is protocol_pdus conn_cfm = {}};
} gate (%q > 0);

trans receiver_idle_2_message_lost
in {
 control: {pid, receiver_idle_2};
}
out {
 control: {pid, receiver_data_rec_ready_0};
};

trans receiver_data_rec_ready_1
in {
 control: {pid, receiver_data_rec_ready_0};
 lossy_channel: {pid, q};
}
out {
 control: {pid, receiver_data_rec_ready_0};
 lossy_channel: {pid, -q};
} gate (/q > 0);
```