Modeling Railway Control Systems in Promela

Roberto Nardone[1](✉), Ugo Gentile[1], Massimo Benerecetti[1], Adriano Peron[1], Valeria Vittorini[1], Stefano Marrone[2], and Nicola Mazzocca[1]

[1] Università di Napoli Federico II, Naples, Italy
{roberto.nardone,ugo.gentile,massimo.benerecetti,adrperon, valeria.vittorini,nicola.mazzocca}@unina.it
[2] Seconda Università di Napoli, Naples, Italy
stefano.marrone@unina2.it

Abstract. This paper presents an approach to systematically build Promela models with the aim of generating test cases within the system level testing process of railway control systems. The paper focuses on the encoding of the system model, of the aspects related to the representation of possible execution environments and their interaction with the system. The input for building a Promela model of the system under test is a state machine based specification. Indeed, state machines are one of the most common notations used in industrial settings to model critical systems and allow for easily obtaining the Promela model of the system by applying a well structured transformational approach; furthermore, state-based formalism are also highly recommended by CENELEC norms to model railway control systems.

In our approach Dynamic State Machines (DSTMs) are used, a newly developed extension of hierarchical state machines which allow for modeling dynamic instantiation of processes. The approach is applied to a functionality of the Radio Block Centre, the vital core of the ERTMS/ETCS Control System, in order to show the feasibility and effectiveness of the generation of the Promela model on a real system.

Keywords: Model checking · Promela · SPIN · Dynamic state machine · CRYSTAL · Railway control systems · Test case generation

1 Introduction and Related Work

The extensive usage of model checking in the Verification&Validation (V&V) activities in the context of control systems development is not a common practice in industry. One of the reasons is the difficulty of building a non trivial model of the system under test (and expressing the properties to be verified) from the artifacts produced during the verification and testing process, without requiring radical changes in the process itself. Other reasons may be the lack of efficiency of the available approaches or the lack of expressive power of the languages used to build the system models. This paper addresses these problems with specific reference to the railway domain. The European norm CENELEC EN50128 [3]

C. Artho and P.C. Ölveczky (Eds.): FTSCS 2015, CCIS 596, pp. 121–136, 2016.
DOI: 10.1007/978-3-319-29510-7_7

emphasizes the usage of model checking as one of the highly recommended techniques to be exploited for formal verification purposes. We propose an automatable approach to build a Promela model, which can be easily integrated into V&V activities. The resulting Promela model can be conceptually divided into two main parts: the first one consists of a set of Promela processes obtained by translating a state-based specification of the system under test (SUT), the second one is a dedicated Promela process modeling possible environment executions. In this work we adopt DSTM (Dynamic State Machine) [9] as the formal language used to model the SUT. DSTM extends Hierarchical State Machine (HSM, [1,8]) and allows for dynamic instantiation of machines (processes), procedure calls, parallelism, parameter passing, interrupts, communication through global variables and channels. The basic ideas underlying the proposed approach are not new. In past work [5] a model-based approach is proposed for the formal verification of the executable code of a railway control system. Several translations from state-based formalisms to model checkers have been proposed in the literature. For example, the work [13] describes an approach to automatically generate test cases for code coverage, by exploiting the capability of the NuSMV model checker. A similar approach is presented in the work [4], which focuses on a methodology to encode Abstract State Machine into Promela, in order to automatically generate test cases. In the past work [12] timing constraints, specified with MARTE Profile, are modeled as automata and then translated into Promela models for the verification of constraints fulfillment.

With respect to the literature, the major strength of our work resides in the definition of a structured approach to build non trivial Promela models taking into account both the issues to be faced in modeling the SUT and the (possibly non-deterministic) behaviors of the environment. The proposed approach is fully automatable starting from a DSTM specification and can be easily integrated in existing industrial settings. The ability of constraining the possible inputs to the SUT provided by the non-deterministic environment, allows, on the one hand, to achieve efficiency in terms of state space generation and analysis effort and, on the other, to prevent the generation of unfeasible test cases.

This paper provides a bird-eye view on the overall modeling approach, in particular a description of how some of the features of DSTM are translated into Promela and the definition of the environment model are presented by using a running example. A complete case study is also proposed, based on a functionality of the Radio Block Centre, a real railway control system. The paper is organized as follows: Sect. 2 summarizes the essential features of DSTM and introduces the running example. Section 3 presents the approach to construct the Promela model. Section 4 contains the railway case study and, finally, Sect. 5 provides some closing remarks and hints about future work.

2 Background and Running Example

DSTM [9] is a newly defined formalism developed within the context of the ongoing ARTEMIS Joint Undertaking project CRitical sYSTem engineering

AcceLeration[1] (CRYSTAL, [10]). It has been designed according to the needs expressed by a railway industry in order to be easily integrated in the testing process of signaling control systems. The ultimate goal is to develop an interoperable testing environment providing a high level of automation [2].

As the aim of the paper is to introduce an approach to build non trivial Promela models starting from a DSTM specification of the SUT, in the following we provide an informal introduction to DSTM by means of the toy running example depicted in Fig. 1. The example contains two machines: a machine modeling a Set-Reset (SR) flip-flop (Fig. 1(a)), and machine that models a 4-bit register (Fig. 1(b)) by activating four parallel instances of the flip-flop machine.

A DSTM is a collection of parametric machines, channels, variables and datatypes. The evolution of a DSTM is a sequence of instantaneous reactions (*steps*). A step is a maximal set of transition firings which are triggered by the current set of available events avoiding sequential firings of transitions within the same step. DSTM allows for the definition of (*internal* or *external*) channels and global variables that allow for communication between machines. Additionally, DSTM gives the possibility to build complex types starting form basic ones. Specifically, basic types are integer, enumerations and channel names. Basic types can be composed to constitute compound types and multi-types. Compound types are structured types similar to records of basic types; multi-types, instead, are collections of basic and compound types. Channels may convey messages of any available type.

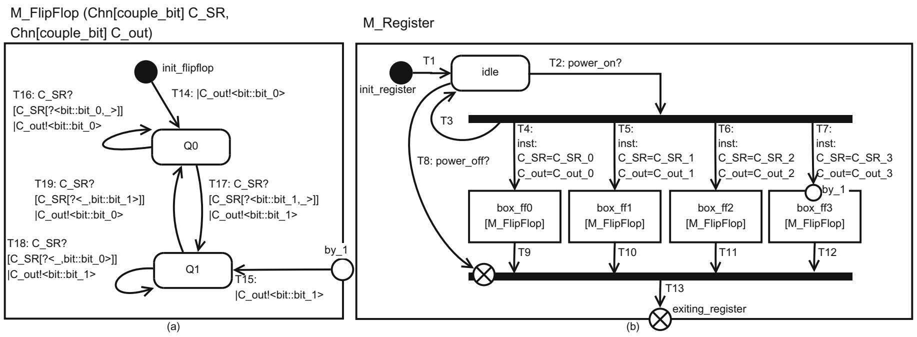

Fig. 1. (a) SR flip-flop model. (b) 4-bit register model

A single machine is composed of vertices, transitions and parameters. Different kinds of vertices may be included in a machine. *Node*s represent the possible control states (e.g., node `idle` of `M_Register` in Fig. 1). An *initial node* is also present in each machine, corresponding to the default entry (e.g., node `init_register` of `M_Register`). Moreover, a machine may contain additional entering nodes (e.g., node `by_1` of `M_FlipFlop`) and exiting nodes (e.g., node

[1] http://www.crystal-artemis.eu/.

`exiting_register` of `M_Register`). *Box*es represent single or multiple machine invocations (parallel procedure calls). A transition entering a box models the invocation of the machine(s) associated with the box, while a transition leaving a box corresponds to a return from that machine(s). For instance, transitions `T4`-`T7` perform invocations of the parametric machine `M_FlipFlop`, with suitable instantiation of its parameters, by entering boxes `box_ff0`, `box_ff1`, `box_ff2` and `box_ff3`, respectively. Parallel behavior can be modeled either by associating multiple machines with a single box, or by explicitly splitting and merging the control flow using fork and join constructs. To this end, *Fork* and *Join* pseudonodes are provided in DSTM. A transition exiting a fork can execute either synchronously or asynchronously with the currently executing process. In the latter case, a transition from the fork node leads to a node of the caller machine. For instance, transition `T2` triggers an asynchronous fork, instantiating four boxes whose associated machines execute asynchronously with the caller machine `M_Register`. *Join* nodes allow for merging of multiple control flows from concurrently executing processes. It either synchronizes the termination of the involved processes or forces their termination if a *preemptive* transition, marked by the symbol $\otimes$, enters the join node. Note that asynchronous forks, occurring within loops, allow for the dynamic instantiation of processes. This feature may lead to an unbounded number of processes and, as a consequence, to an unbounded state space. To allow Spin to analyze of the resulting designs, the generation of the corresponding Promela models bounds the number of instantiation of each machine.

Transitions are decorated with *trigger*s, *condition*s and *action*s. With reference to Fig. 1, the decoration of transitions `T2` in machine `M_Register` contains only a trigger which tests the presence of a message on the channel `power_on`, no additional condition is required for firing and no action is performed. Transition `T16` in machine `M_FlipFlop`, instead, requires the presence of a specific message on the parametric channel `C_SR`. In fact, messages sent over the channel `C_SR` are structured as `couple_bit` $=$ $\langle$`bit,bit`$\rangle$, where `bit` is an enumerative type `bit` $=$ $\{$`bit_0,bit_1`$\}$. The trigger of `T16` tests the presence of a message over the channel `C_SR`, while the condition further requires that the content of the received message is a pair whose first component has the value `bit::bit_0`, whereas the second component is simply ignored (denoted by the wildcard "_"). The action performed upon firing of the transition corresponds to the delivery of the value `bit::bit_0` on the parametric channel `C_out`. Parameters associated to a machine (e.g., parameters `C_SR` and `C_out` of machine `M_FlipFlop`) are instantiated at invocation time.

3 Definition of the Promela Model

Starting from a DSTM specification of the SUT, we build a Promela model with the goal of generating test sequences, exploiting the capability of model checkers to build counterexamples of violated properties. By following a structured approach, a set of Promela processes is systematically built from the DSTM model of the SUT, together with a Promela process modeling a (non-deterministic) environment. We exemplify the generation of test sequences by assuming that the coverage of transitions is required. A more general discussion about requirements and how they are expressed is out of the scope of this work. Before describing how the Promela model can be automatically built, we provide a high level overview of the steps implementing the translation of DTSMs into Promela.

3.1 Generation Steps

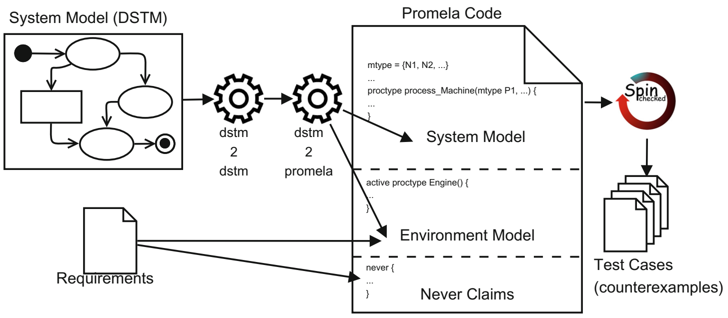

Fig. 2. Overview of the general approach.

Figure 2 depicts the steps which enable the automatic generation of the Promela model. Since Promela does not support hierarchical specifications, the encoding of a hierarchical DSTM model is performed in two phases. The first one is a *dstm2dstm*, where the hierarchical source model is flattened, to get rid of the hierarchical structure of a DSTM (the vertical modularity) and suitably encoding the horizontal modularity (i.e., parallelism). In this phase, all boxes, forks and joins are removed from the model and additional nodes are inserted when necessary. The dynamic instantiation of machines is translated into specific *run* commands, added to the action list of those transitions replacing the ones entering the boxes. During the second step, named *dstm2promela*, the flat DSTM model resulting from the previous phase is translated directly into Promela and a dedicated Promela process, called `Engine`, is generated to model the execution environment of the SUT.

The test goal and the assumptions about the execution environment of the SUT (if any) provide requirements that the model of the environment must satisfy. These requirements drive: (1) the refinement of the *Engine* process, by inserting constraints on the communication channels, and (2) the generation of *never claims*. Never claims are used to focus the analysis of the system to behaviors of special interest. In our case, never claims are used to describe those behaviors of the system that exhibit the occurrence of some desired transition. When a never claim is fulfilled, the model checker Spin provides a counterexample, which, in the present setting, corresponds to a possible test sequence that covers the specified transition.

3.2 Building the System Model

The step semantics of DSTM prevents sequential firings of transitions within the same execution step. Hence, we need to guarantee that at most one enabled transition can fire for each active process. To this aim, each DSTM machine M is translated into a Promela process, called `process_M`, which is instantiated by its caller process using the Promela command *run*, which allows for dynamic activation of processes. Each process encoding machine M is then executed until a termination message, sent by its caller, is received over a special channel `ch_term_M` (defined as a local channel to the caller). Each Promela process, modeling a DSTM machine, must own a token in order to fire an enabled transition. When a process owns a token it is scheduled, it consumes its token and: if none of its transitions is enabled, the process propagates the token to each of the child processes it has previously activated, if any; otherwise, one of the enabled transitions is selected and executed. At the beginning of each step only the process corresponding to the initial machine (main) owns the token. This machine is `M_Register` in the case of the running example. A set of global variables and channels is declared for each process. In Listing 1 all the automatically generated declarations for the running example are reported.

Listing 1. Global variables for the running example.

```
1. #define MAX_PROC 6
2. mtype last_transition;
3. bit has_token[MAX_PROC];
4. mtype = {init_register , idle , exiting_register};
5. mtype = {T1, T2_T3_T4_T5_T6_T7, T8_T9_T10_T11_T12_T13};
6. mtype = {init_flipflop , by_1 , Q0, Q1};
7. mtype = {T14, T15, T16, T17, T18, T19};
8. mtype state_M_Register[MAX_PROC];
9. mtype transition_M_Register[MAX_PROC];
10. mtype state_M_FlipFlop[MAX_PROC];
11. mtype transition_M_FlipFlop[MAX_PROC];
12. mtype = {bit_0 , bit_1 };
13. chan power_on = [2] of {bit};
14. chan power_off = [2] of {bit};
15. chan C_SR_0 = [2] of {mtype, mtype};
16. chan C_out_0 = [2] of {mtype};
17. chan C_SR_1 = [2] of {mtype, mtype};
18. chan C_out_1 = [2] of {mtype};
19. chan C_SR_2 = [2] of {mtype, mtype};
20. chan C_out_2 = [2] of {mtype};
21. chan C_SR_3 = [2] of {mtype, mtype};
22. chan C_out_3 = [2] of {mtype};
```

A global variable `has_token`, typed as a bit array, is used to store the assignment of the tokens described before (line 3). Specifically, this array contains 1 in the i-th location if the machine with *pid* equal to i currently has the token. Note that this array is global and visible to the entire Promela model. Two enumeration types (*mtype*) introduce symbolic names for nodes and transitions (e.g., lines 6 and 7 correspond to `process_M_FlipFlop`). The *mtype* vector variable `state_M` is used to maintain the current states of all the instances of machine `M` (e.g., line 10). Its elements are all initialized to the initial node of the corresponding machine. A *mtype* vector variable `transition_M` is used to keep track, for each instance of machine `M`, of the transition that fires in the current step (e.g., line 11). From lines 12 to 22 types and channels are declared. Both channels `C_SR` and `C_out` can store two messages, in order to correctly implement the step semantics, as it will be explained in Sect. 3.3. Finally, the variable `last_transition` (line 2) is used to store the name of the last transition covered in an execution. This information is used for instrumentation purposes, specifically it allows for the definition of the never claim requiring to find a the covering of a transition.

Promela Model of Machine *M_FlipFlop.* Each node of a machine is mapped into a *guarded statement* of the form *guard −>statement*. In Listing 2 an excerpt of the Promela translation of machine `M_FlipFlop` is reported. Notice that the channel names `C_SR` and `C_out` are parameters. The actual names are provided by the caller process, which can distinguish the different instances of `M_FlipFlop`. Furthermore, the `ch_term` channel is added as parameter: over this channel, defined locally to the caller, this latter sends the termination message to the one executing.

Listing 2. Promela code of machine `M_FlipFlop` (excerpt).

```
24. proctype process_M_FlipFlop(chan C_SR; chan C_out, chan ch_term) {
25.   do

46.   :: (state_M_FlipFlop[_pid]==Q0 && has_token[_pid]==1) ->
47.     atomic {
48.       printf("<current node[\%d] = Q0>\n",_pid);
49.       has_token[_pid]=0;
50.       if
51.       :: (C_SR?[bit_0,_]) ->
52.         C_out!bit_0;
53.         printf("<firing transition[\%d] = T16>\n",_pid);
54.         transition_M_FlipFlop[_pid]=T16;
55.         state_M_FlipFlop[_pid]=Q0;
56.         last_transition=T16;
57.       :: (C_SR?[bit_1,_]) ->
58.         C_out!bit_1;
59.         printf("<firing transition[\%d] = T17>\n",_pid);
60.         transition_M_FlipFlop[_pid]=T17;
61.         state_M_FlipFlop[_pid]=Q1;
62.         last_transition=T17;
63.       :: else
64.       fi;
65.     }

86.   od unless {
87.     ch_term?1;
88.     printf("<Machine M_FlipFlop[\%d] terminated>\n",_pid);
89.   }
92. }
```

The *guard* of the statement checks whether some enabled transition is allowed to fire from a node: the current node must be the source node of the transition (e.g., `state_M_FlipFlop[_pid]==Q0` in Listing 2, line 45) and the process owns the token (`has_token[_pid]==1`, line 45). The statement is *atomic* and contains a sequence of statements executed indivisibly. The first statement in the sequence consumes the token. Then, a conditional statement contains one guarded statements for each transition exiting from that node (e.g., lines 51, 57). Their guards correspond to the enabling condition of the DSTM transitions and the associated statements translate the actions of the transitions. The actions (if any) associated with a DSTM transition are translated into basic Promela statements and operators, and they are executed when the associated guarded statement is selected. If more than one guarded statement is executable, one of them is non-deterministically selected. The *else* branch in the conditional statement (e.g., line 63) is taken when no transition can fire. The process is executed until a termination message, sent by its caller, is received by the caller over the channel `ch_term` (lines 86–89).

Promela Model of Machine `M_Register`. As opposed to the previous component, machine `M_Register` has a hierarchical structure, which requires to be flattened before translating it to Promela. The flattening phase removes all the boxes, fork and join constructs. In doing that, some transitions may be modified, eliminated or added. Moreover, additional variables and channels are introduced and proper conditions and actions are modified or added to the decorations of existing transitions. These elements are used to provide additional information and directives for the generation of the Promela code. The resulting flattened model of the DSTM in Fig. 1(b) is reported in Fig. 3.

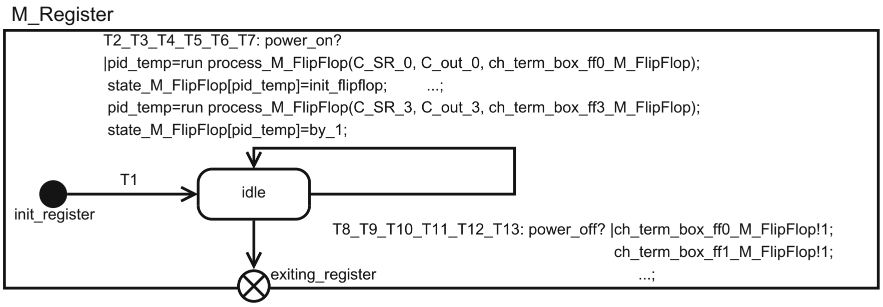
M_Register
T2_T3_T4_T5_T6_T7: power_on?
|pid_temp=run process_M_FlipFlop(C_SR_0, C_out_0, ch_term_box_ff0_M_FlipFlop);
state_M_FlipFlop[pid_temp]=init_flipflop; ...;
pid_temp=run process_M_FlipFlop(C_SR_3, C_out_3, ch_term_box_ff3_M_FlipFlop);
state_M_FlipFlop[pid_temp]=by_1;
T1
idle
init_register
T8_T9_T10_T11_T12_T13: power_off? |ch_term_box_ff0_M_FlipFlop!1;
ch_term_box_ff1_M_FlipFlop!1;
...;
exiting_register

Fig. 3. Promela representation of the hierarchical machine `M_Register`.

The flattening of machine `M_Register` proceeds as follows. Since transition `T2` (see Fig. 1(b)) enters a fork, the process continues its execution after the fork is performed. The DSTM model of `M_Register` is changed as follows:

– the fork and join nodes and the boxes are removed together with their entering and exiting transitions;
– a loop transition from the node `idle` is created which replaces transition `T2`, `T3`, `T4`, `T5`, `T6` and `T7`. The decoration of this transition specifies the trigger of `T2` (i.e., `power_on?`) and actions which contain all the information needed to instantiate the machines called inside the boxes; in particular, the actions specify the Promela statements to execute the four instances of the processes called by the fork operation (e.g., `run process_M_flipflop`) and set their initial state (e.g., `state_M_flipflop[pid_temp]=initial`).
– a transition from node `idle` to the exit node is created which replaces transitions `T8`, `T9`, `T10`, `T11`, `T12` and `T13`. The decoration of this transition specifies the trigger of the preemptive transition `T8` (`power_off?`) and actions encoding the preemptive join by requesting the termination of the four flip-flop processes through message on the termination channels (e.g., `ch_term_box_ff0_M_FlipFlop!1`) which are added to the model.

Listing 3 shows an excerpt of the Promela process encoding the flat machine depicted in Fig. 3 obtained by applying the technique explained above.

Listing 3. Promela code of machine `M_Register` (excerpt).

```
proctype process_M_Register() {
  byte i;
  pid pid_temp;
  bit my_children[MAX_PROC];
  chan ch_term_box_ff0_M_FlipFlop, ch_term_box_ff1_M_FlipFlop,
 ch_term_box_ff2_M_FlipFlop, ch_term_box_ff3_M_FlipFlop;
  do
...
    :: (state_M_Register[_pid]==idle && has_token[_pid]==1) ->
       atomic {
          printf("<current node[\%d] = idle>\n",_pid);
          has_token[_pid]=0;
          if
          :: (power_on?[1]) ->
             pid_temp = run process_M_FlipFlop(C_SR_0, C_out_0,
 ch_term_box_ff0_M_FlipFlop);
             state_M_FlipFlop[pid_temp]=init_flipflop;
             my_children[pid_temp] = 1;
             has_token[pid_temp]=1;

          :: (power_off?[1]) ->
             ch_term_box_ff0_M_FlipFlop!1;
             ch_term_box_ff1_M_FlipFlop!1;
             ch_term_box_ff2_M_FlipFlop!1;
             ch_term_box_ff3_M_FlipFlop!1;
             printf("<firing transition[\%d] = T8_T9_T10_T11_T12_T13}>\n",_pid);
             transition_M_Register[_pid]=T8_T9_T10_T11_T12_T13;
             state_M_Register[_pid]=exiting_register;
             last_transition=T8_T9_T10_T11_T12_T13;
          :: else ->
             for (i : 0 .. MAX_PROC-1) {
                has_token[i]=my_children[i];
             }
          fi;
       }
...
   od unless {
       ch_term_M_Register?[1];
       printf("<Machine process_M_Register[\%d] terminated>\n",_pid);
   }
}
```

3.3 Modeling the Environment

As anticipated in the previous section, the possible environments of the SUT are modeled by a Promela process named `Engine`. This is the first process to be activated in the and it is the only process required to be running in the initial state by using the prefix `active` in its `proctype` declaration. The process `Engine` is in charge of: (1) instantiating the main machine of the system model; (2) non-deterministically generating messages, delivered by the environment on the external channels at the beginning of each execution step; (3) assigning the token to the main machine, starting the execution of a new execution step.

The `Engine` process is activated whenever no statement is executable in any process belonging to the system model. This situation is captured by the `timeout` Promela variable being true. This happens when each process belonging to the system model has *consumed* its own token, meaning that the execution current step is completed. Furthermore, `Engine` uses local variables to non-deterministically generate new messages in the external channels. These local variables are in correspondence to the fields of the compound types exchanged over those channels. Hence, in the running example, we have two variables, `temp1` and `temp2`, as declared at line 162 of the snippet of code reported in Listing 4.

Listing 4. Promela code of machine `Engine` - initialization

```
161. active proctype Engine() {
162.   mtype temp1, temp2;
163.   pid pid_main;
164.   pid_main = run process_M_Register
     ();
165.   state_M_Register[pid_main]=
     init_register;
166.   printf(   <ENGINE: main machine
     has pid = %d\n>", pid_main);
167.   //Generation of first message
168.   power_on!0;
169.   power_off!0;
170.   C_SR_0!bit_0, bit_0;
171.   C_out_0!bit_0;
```

Listing 5. Promela code of machine `Engine` - generation of new messages

```
178. generation:
179.   atomic {
180.     printf("<ENGINE: message
     generation>\n");
181.     //MESSAGES ON power_on
182.     if
183.     :: (len(power_on)==1) ->
184.       if
185.       :: (1) -> temp1=0;
186.       :: (1) -> temp1=1;
187.       fi;
188.       printf("<ENGINE: power_on -
     generated <%d>>\n", temp1);
189.       power_on!temp1;
190.     :: (full(power_on)) -> skip;
191.     fi;
192.     power_on?temp1;
...
343.     //GIVE TOKEN TO THE MAIN
    PROCESS
344.     has_token[pid_main] = 1;
345.     printf("<ENGINE: end execution
    >\n");
346.   }
347. do
348.   :: timeout ->
349.     goto generation;
350. od
351. abort:
352.   skip;
353. }
```

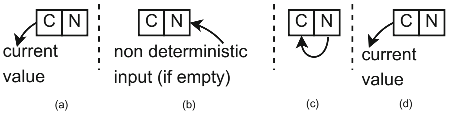

Fig. 4. Message generation for the `power_on` channel.

First, the `Engine` process runs the main machine (i.e., `process_M_Register()`) and stores its *pid* in the local variable `pid_main` (Listing 4, line 164). Then, `Engine` initializes the channels (Listing 4, lines 167–171).

After those initialization steps, process `Engine` starts an *atomic* block in which it non-deterministically generates the messages to be sent over the channels (e.g., Listing 5, lines 182–187 initialize the `power_on` channel). The starting statement of this block is identified by the label `generation` at line 178 in Listing 5.

The subsequent evolution of the processes is driven by a suitable message handling mechanism, implemented as explained in the following. Each external channel has a buffer that stores two messages (Fig. 4). The first position `C` is used to store the message available in the current step, whereas the position `N` is used to store the message to be delivered in the next step (if any). During the execution of the current step, the processes modeling the SUT can read messages contained in positions `C` of any channels, without removing them. If a new message is produced by the SUT, it is stored in positions `N` of the corresponding channel (Fig. 4(a)). At the beginning of the next step, the `Engine` checks for the

Listing 6. Promela code of machine `Engine` - constraints.

```
212. //MESSAGES ON C_SR_0
213. if
214. :: (len(C_SR_0)==1) ->
215.   if
216.   :: (1) -> temp1=0;
217.   :: (1) -> temp1=1;
218.   fi;
219.   if
220.   :: (1) -> temp2=0;
221.   :: (1) -> temp2=1;
222.   fi;
223.   if
224.   :: (temp1==1 && temp2==1)-> goto abort;
225.   :: else -> skip;
226.   fi;
```

presence of messages in positions `N`, (Listing 5, line 183). If a position `N` does not contain a message generated by the SUT processes during the previous step, the `Engine` generates a new message by using `temp` variables and `if` statements (Fig. 4(b), Listing 5, lines 184–189). Finally, the `Engine` consumes all the messages contained in positions `C` (line 192), by moving the content of position `N` in each channel to position `C`, thus making the messages previously generated for the next steps available (Fig. 4(c)).

Note that the *receive* statement at line 192 is always executable. This is ensured by the fact that the SUT never removes messages from the external channels. Therefore, that statement is never blocked, as two messages are always stored in each channel when it is executed.

The `generation` block ends by assigning the token to the main process (Listing 5, line 344). Then, the `Engine` process enters the `do` construct, where it waits until the Promela global variable `timeout` evaluates to true. This happens when no statement is executable in the active processes, hence when all the SUT processes have consumed their token. In this case, `Engine` executes a jump to the `generation` label, starting a new step.

3.4 Constraining Behaviors

The non-deterministic generation of messages to be sent over the channels can be constrained to a set of requirements that the desired environment must fulfill. Such constraints can be used to prevent the environment to prompt the system with unfeasible combinations of inputs.

The simplest constraint a designer may require is to avoid the generation of conflicting messages over the channels. As an example, the SR flip-flop cannot be prompted with both $R = 1$ and $S = 1$. This constraint can be expressed in Promela as shown in Listing 6. The constraint is included in the already described `generation` block for channel `C_SR_0`. After the generation of the values for the signals S and R in the variables `temp_1` and `temp_2` respectively, the statements reported at lines 223–225 check that these values are not both equal to 1. If the constraint is not satisfied, the `Engine` process jumps to the `abort` label (reported in Listing 5), which immediately ends this process, interrupting the related behavior. Note that the alternative handling of constraint violations that generates a new set of values for the messages is not an efficient solution, since it increases the number of possible execution paths in the state space, without adding meaningful behaviors. Similarly, we can express constraints involving different fields of the same compound message and constraining the generation of messages subject to the occurrence of specific events. These kinds of constraints are not described here for sake of space.

4 A Case Study in the Railway Domain

ERTMS/ETCS (European Rail Traffic Management System/European Train Control System, [11]) is a standard for the interoperability of the European railway signalling systems ensuring both technological compatibility among trans-European railway networks and integration of the new signalling system with the existing national interlocking systems. The ERTMS/ETCS specification identifies three functional levels featuring growing complexity. They can be implemented singularly or in conjunction and mainly differ in the communication mechanisms adopted to control the trains. Level 2 and Level 3 represent two more cutting-edge solutions than Level 1, at this moment Level 2 is the most widespread choice between Level 2 and Level 3. A reference architecture for ERTMS/ETCS systems consists of three main subsystems: the on-board system is the core of the control activities located on the train; the line side subsystem is responsible for providing geographical position information to the on-board subsystem; the trackside sub-system is in charge of monitoring the movement of the trains. The Radio Block Centre (RBC) is the most important component of the track side subsystem of the ERTMS/ETCS architecture. RBC is a computing system whose aim is to guarantee a safe inter-train distance on the track area under its supervision. It interacts with the on-board system by managing a communication session, by using the EURORADIO protocol and the GSM-R network. In the following, part of a realistic realization of an RBC procedure is described, together with the test generation procedure that demonstrates how the proposed approach can be effectively applied to obtain test sequences.

4.1 The Communication Procedure of the Radio Block Centre

The Communication procedure is modeled by the DSTM specification shown in Fig. 5. The main machine **M_CommunicationEstablishment** (Fig. 5(a)) is in charge of modeling the management of the connection requests issued by the trains. It accepts a limited number of requests (collected in variable `V_cont`) and for each accepted request it instantiates a new machine `M_ManageTrain` by entering the box `MCE_manageTrain`. Three transitions exit from node `MCE_idle`: `MCE_T03`, `MCE_T06` and `MCE_T02`. `MCE_T03` enters the fork, it is triggered by the availability of a message on channel `C_request` and it is guarded by the condition `V_cont<=3`. The action of this transition delivers acceptance message over channel `C_answer`, increments counter `V_cont` and stores in variables `V_chSystemVersion`, `V_chAck` and `V_chSessionEstablished` the names of the channels to be used to communicate with the train. The asynchronous control flow exiting from this fork returns back to node `MCE_idle`. When an instance of machine `M_ManageTrain` terminates its execution, transitions `MCE_T06` and `MCE_T07` merge the control flow by entering the join node; transition `MCE_T08` exiting the join decrements the counter `V_cont`. Transition `MCE_T02` from node `MCE_idle`, instead, is activated on receiving a connection request, when the maximal number of service requests has been reached. The action of this transition delivers of a suitable refusal message over channel `C_answer`. Machine `M_ManageTrain` (Fig. 5(b)) models the management of the communication procedure with a specific train. It takes the names of the channels, on which the train and RBC will communicate, as parameters. This machine enters node `MMT_idle` and then instantiates machine `M_SessionEstablishment`, which models the session establishment protocol, by entering the corresponding box. Machine `M_SessionEstablishment` (Fig. 5(c)) can terminate its execution with different exiting conditions (i.e., different exiting

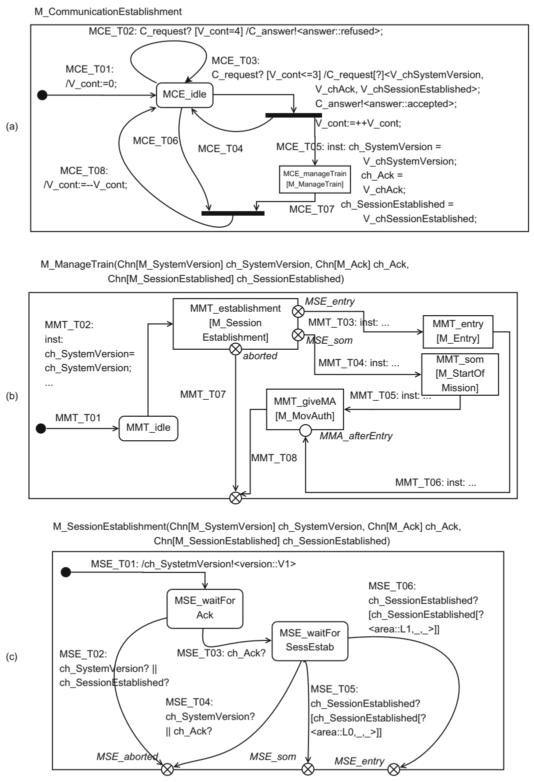
M_CommunicationEstablishment
MCE_T02: C_request? [V_cont=4] /C_answer!<answer::refused>;
MCE_T01: /V_cont:=0;
MCE_T03: C_request? [V_cont<=3] /C_request[?]<V_chSystemVersion, V_chAck, V_chSessionEstablished>; C_answer!<answer::accepted>;
MCE_idle
V_cont:=++V_cont;
(a)
MCE_T06
MCE_T04
MCE_T05: inst: ch_SystemVersion = V_chSystemVersion; ch_Ack = V_chAck; ch_SessionEstablished = V_chSessionEstablished;
MCE_T08: /V_cont:=--V_cont;
MCE_manageTrain [M_ManageTrain]
MCE_T07
M_ManageTrain(Chn[M_SystemVersion] ch_SystemVersion, Chn[M_Ack] ch_Ack, Chn[M_SessionEstablished] ch_SessionEstablished)
MMT_establishment [M_Session Establishment]
MSE_entry
MMT_T03: inst: ...
MMT_entry [M_Entry]
MMT_T02: inst: ch_SystemVersion= ch_SystemVersion; ...
MSE_som
aborted
MMT_T04: inst: ...
MMT_som [M_StartOf Mission]
(b)
MMT_T07
MMT_giveMA [M_MovAuth]
MMT_T05: inst: ...
MMT_T01
MMT_idle
MMA_afterEntry
MMT_T08
MMT_T06: inst: ...
M_SessionEstablishment(Chn[M_SystemVersion] ch_SystemVersion, Chn[M_Ack] ch_Ack, Chn[M_SessionEstablished] ch_SessionEstablished)
MSE_T01: /ch_SystetmVersion!<version::V1>
MSE_waitFor Ack
MSE_T06: ch_SessionEstablished? [ch_SessionEstablished[? <area::L1,_,_>]]
MSE_waitFor SessEstab
(c)
MSE_T02: ch_SystemVersion? || ch_SessionEstablished?
MSE_T03: ch_Ack?
MSE_T04: ch_SystemVersion? || ch_Ack?
MSE_T05: ch_SessionEstablished? [ch_SessionEstablished[? <area::L0,_,_>]]
MSE_aborted
MSE_som
MSE_entry

Fig. 5. DSTM model of the Communication Procedure.

nodes). If the communication session with the train has been successfully established, then the machine exits via either via node `MSE_som` or node `MSE_entry`, according to the specific communication mode established. Depending on the exit node of this machine, machine `M_ManageTrain` then instantiates either `M_StartOfMission` or machine `M_Entry`. If, on the other hand, the session establishment protocol aborts, then it terminates its execution in node `MSE_aborted`. Finally, machine `M_MovAuth` is instantiated after the termination of either one of mechines `M_StartOfMission` and `M_Entry`, which provides the train with the Movement Authority.

4.2 Results

The construction of the Promela model is automatically generated as explained in Sect. 3. The model contains as many processes as DSTM machines depicted in Fig. 5 plus the *Engine* process. As the structure of the Promela model is exactly the same of the code discussed in Sects. 3.2 and 3.3, only a portion of the code for **process_M_CommunicationEstablishment** is shown in Listing 7. The entire model of the case study contains around 1250 lines of code, where the first 75 of them are types and variable declarations.

Listing 7. Promela model of $M_Communication Establishment()$

```
proctype process_M_CommunicationEstablishment(chan ch_term) {
  byte i;
  pid pid_temp;
  bit my_children[MAX_PROC];
  chan ch_term_MCE_manageTrain_M_ManageTrain,
       ch_term_MCE_manageTrain_M_ManageTrain_exiting;
  do
  :: (state_M_CommunicationEstablishment[_pid]==MCE_initial && has_token[
       _pid]==1) ->
  atomic {
    printf("<current node[
    has_token[_pid]=0;
    V_cont=0;
    printf("<firing transition[
    transition_M_CommunicationEstablishment[_pid]=MCE_T01;
    state_M_CommunicationEstablishment[_pid]=MCE_idle;
    last_transition=MCE_T01;
  }
  :: (state_M_CommunicationEstablishment[_pid]==MCE_idle && has_token[_pid
       ]==1) ->
  atomic {
    printf("<current node[
    has_token[_pid]=0;
    if
    :: (C_request?[_,_,_] && V_cont==4) ->
      C_answer!refused;
      printf("<firing transition[
      transition_M_CommunicationEstablishment[_pid]=MCE_T02;
      state_M_CommunicationEstablishment[_pid]=MCE_idle;
      last_transition=MCE_T02;
    :: (C_request?[_,_,_] && V_cont<=3) ->
      C_request?V_chSystemVersion,V_chAck,V_chSessionEstablished;
      C_answer!accepted;
      V_cont++;
      pid_temp=run process_M_ManageTrain(V_chSystemVersion,V_chAck,
           V_chSessionEstablished, ch_term_MCE_manageTrain_M_ManageTrain,
           ch_term_MCE_manageTrain_M_ManageTrain_exiting);
      state_M_ManageTrain[pid_temp]=MMT_initial;
      my_children[pid_temp]=1;
      has_token[pid_temp]=1;
      printf("<firing transition[
      transition_M_CommunicationEstablishment[_pid]=MCE_T03_MCE_T04_MCE_T05;
      state_M_CommunicationEstablishment[_pid]=MCE_idle;
      last_transition=MCE_T03_MCE_T04_MCE_T05;
    :: (ch_term_MCE_manageTrain_M_ManageTrain_exiting?[1]) ->
      ch_term_MCE_manageTrain_M_ManageTrain_exiting?_;
      ch_term_MCE_manageTrain_M_ManageTrain!1;
      V_cont--;
      printf("<firing transition[
      transition_M_CommunicationEstablishment[_pid]=MCE_T06_MCE_T07_MCE_T08;
      state_M_CommunicationEstablishment[_pid]=MCE_idle;
      last_transition=MCE_T06_MCE_T07_MCE_T08;
    :: else ->
      for (i : 0 .. MAX_PROC-1) {
        has_token[i]=my_children[i];
      }
    fi;
}
od unless {
    ch_term?1;
    printf("<Machine M_CommunicationEstablishment[
}
}
```

In order to show the effectiveness of the approach, we report the resulting performance of the Promela model for generating a test sequence that covers transition `MSE_T06` of machine `M_SessionEstablishment`. The test sequence is obtained by generating a Promela never claim that checks for the existence of behaviors in which transition `MSE_T06` is taken (i.e., such that the condition `last_transition==MSE_T06` holds). The corresponding never claim is shown in Listing 8. This Promela model has been executed by SPIN [6] on a personal computer equipped with an Intel Core-i7, 8GB of RAM. The generation of the test sequence requires the exploration of **5211 states** analyzed in **0.234 s**.

Listing 8. Never claim

```
never {
step1:
  if
    :: (last_transition==MSE_T06) -> goto endStep
    :: else -> goto step1
  fi;
endStep: skip
}
```

5 Conclusions and Future Work

In this paper we presented a fully automatable approach to build a non trivial Promela model from a DSTM specification of a system under test. The approach has been defined to be integrated into existing testing environments in railway industrial settings and provide practical means to support the automatic generation of test sequences for gray-box testing of control systems. We are currently completing the process for the automatic translation of DSTM models into Promela and the construction of the Promela model modeling the environment of the SUT. This involves the implementation of a chain of model transformations partially written in ATL [7]. More work along several directions is needed to provide a complete test case generation environment. In particular, we are currently working on automating the construction of test specifications to obtain transition coverage, on optimizing the generation of the test cases, and on providing the end-user with a proper presentation of the generated sequences.

Acknowledgments. This paper is partially supported by research project CRYSTAL (Critical System Engineering Acceleration), funded from the ARTEMIS Joint Undertaking under grant agreement no. 332830 and from ARTEMIS member states Austria, Belgium, Czech Republic, France, Germany, Italy, Netherlands, Spain, Sweden, United Kingdom.

References

1. Alur, R., Kannan, S., Yannakakis, M.: Communicating hierarchical state machines. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 169–178. Springer, Heidelberg (1999)

2. Di Martino, B., et al.: An interoperable testing environment for ERTMS/ETCS control systems. In: Bondavalli, A., Ceccarelli, A., Ortmeier, F. (eds.) SAFECOMP 2014. LNCS, vol. 8696, pp. 147–156. Springer, Heidelberg (2014)
3. CENELEC EN50128: communication, signalling and processing systems - software for railway control and protection systems (2011)
4. Riccobene, E., Rinzivillo, S., Gargantini, A.: Using spin to generate testsfrom ASM specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 263–277. Springer, Heidelberg (2003)
5. Haxthausen, A.E., Peleska, J., Kinder, S.: A formal approach for the construction and verification of railway control systems. Formal Aspects Comput. **23**(2), 191–219 (2011)
6. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual, vol. 1003. Addison-Wesley, Reading (2004)
7. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: a model transformation tool. Sci. Comput. Program. **72**(1), 31–39 (2008)
8. Lanotte, R., Maggiolo-Schettini, A., Peron, A., Tini, S.: Dynamic hierarchical machines. Fundam. Inf. **54**(2–3), 237–252 (2002)
9. Nardone, R., et al.: Dynamic state machines for formalizing railway control system specifications. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2014. CCIS, vol. 476, pp. 93–109. Springer, Heidelberg (2015)
10. Pflügl, H., El-Salloum, C., Kundner, I.: Crystal, critical system engineering acceleration, a truly european dimension. ARTEMIS Mag. **14**, 12–15 (2013)
11. UIC. ERTMS/ETCS class1 system requirements specification, ref. SUBSET-026, issue 2.2.2 (2002)
12. Yin, L., Mallet, F., Liu, J.: Verification of marte/ccsl time requirements in promela/spin. In: 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 65–74 (2011)
13. Zheng, Y., Zhou, J., Krause, P.: A model checking based test case generation framework for web services. In: Fourth International Conference on Information Technology, ITNG 2007, pp. 715–722. IEEE (2007)