




From Dynamic State Machines to Promela

Massimo Benerecetti¹, Ugo Gentile², Stefano Marrone³, Roberto Nardone⁴,
Adriano Peron¹, Luigi L. L. Starace¹, and Valeria Vittorini¹

¹ University of Naples Federico II, Naples, Italy
{massimo.benerecetti,adriano.peron2,valeria.vittorini}@unina.it,
luigi.starace@gmail.com

² CERN, Geneva, Switzerland
ugo.gentile@cern.ch

³ Università della Campania “Luigi Vanvitelli”, Caserta, Italy
stefano.marrone@unicampania.it

⁴ University Mediterranea of Reggio Calabria, Reggio Calabria, Italy
roberto.nardone@unirc.it

Abstract. Dynamic State Machines (DSTM) is an extension of Hierarchical State Machines recently introduced to answer some concerns raised by model-based validation of railway control systems. However, DSTM can be used to model a wide class of systems for design, verification and validation purposes. Its main characteristics are the dynamic instantiation of parametric machines and the definition of complex data types. In addition, DSTM allows for recursion and preemptive termination. In this paper we present a translation of DSTM models in Promela that can enable automatic test case generation via model checking and, at least in principle, system verification. We illustrate the main steps of the translation process and the obtained Promela encoding.

1 Introduction

Dynamic State Machine (DSTM) is a recently-developed modelling language [1], developed in the context of the ARTEMIS Joint Undertaking project CRYSTAL (CRITICAL SYSTEM engineering ACCELERATION) [10]. DSTM has been devised to explicitly meet industrial requirements in design, verification and validation of complex control systems, and includes in its formal framework both complex control flow constructs (such as asynchronous forks, preemptive termination, recursive execution) and complex data flow constructs (such as custom complex type definition, parametric machines, and inter-process communication). DSTM borrows many syntactic elements from UML Statecharts, and extends them with the notion of module and with the possibility of recursion and dynamic instantiation. The possibility of modelling both complex behaviours and data enables the usage of DSTM at different levels of abstraction and for different purposes, for example property verification and model-based testing.

The ultimate objective of ongoing work on DSTM is to enable its usage within model-driven tool chains for application or product life-cycle management. In this direction, this paper presents a transformation from DSTM to

PROMELA in order to provide the necessary support for the automatic integration of verification and validation methodologies based on DSTM into industrial verification and validation processes. Previous work [1, 8, 9] provides the motivation for the introduction of DSTM, the formal definition of the syntax semantics, and its application to the validation of railway control systems. In those papers the encoding of DSTM models to PROMELA models was merely sketched. Here a complete translation is presented, emphasizing how the hierarchical structure of DSTM models can be encoded into the modular features of PROMELA embodied in the notion of process types. Even though, due to space constraints, a precise formal account of the equivalence between the DSTM semantics presented in [1] and the resulting PROMELA encoding is not provided here, the correctness can be stated in terms of a suitable correspondence between executions of a DSTM and of its PROMELA encoding, based on the step semantics. Such a correspondence would allow for formal verification of linear time properties of DSTM with SPIN.

Many works introducing transformations from high level specification languages to formal languages are discussed in the literature. Among them transformations from AADL, MARTE and MARTE-DAM UML profiles are of special interest for the analysis of critical systems (e.g., in [2, 3, 11]). In particular, a transformation from SysML to the specification languages of Spin, Prism and NuSMV model checkers is presented in [5]. The implementation of Statecharts in PROMELA has been studied since 1998 [7]. In [4] an algorithm to automatically encode an ASM specification in PROMELA is presented with the aim of automated generation of test sequences. The novelty of our work is the source formalism and its peculiarities, in particular recursion and dynamic instantiation that are not allowed in other state-based languages.

The paper is organized as follows. Section 2 provides the basics on DSTM and describes some original modelling examples. The translation from DSTM models to PROMELA models is introduced in Sect. 3, where the key issues to be addressed and the adopted solutions are discussed. Section 4 contains some closing remarks and suggestions for future work.

2 Dynamic State Machines

In this section we provide an overview of DSTM through some examples, in order to introduce the main notions used in the rest of the paper. For a complete account of the formal syntax and semantics of DSTM we refer to [1].

A Dynamic State Machine (DSTM) model is a sequence of machines M_1, M_2, \dots, M_n communicating over a set X of global variables and a set C of global communication channels. Machine M_1 is the *initial machine*, namely the highest level of the hierarchical system. Each machine M_i , with $i \in \{2, \dots, n\}$, may be parametric over a set of parameters $P_i \subseteq P$. Parameters are aliases for channels and variables names and are actualized at runtime, when the machine is instantiated, allowing multiple instantiations of the same machine with different parameter values. When a parametric machine is instantiated, each parameter is

mapped to its actual value by means of a *parameter-substitution function*, which associates the parameters with actual ground values. A machine M_i represents a module in a DSTM specification and is defined as a state-transition diagram, whose possible kinds of vertices are:

- node:** basic control state of a machine;
- entering node:** initial pseudo-node of a machine. A machine may specify multiple entering nodes, corresponding to different initial conditions;
- initial node:** default entering pseudo-node of a machine, to be used when no entering node is explicitly specified;
- exit node:** final (or exiting) node of a machine corresponding to different termination conditions;
- box:** node modelling the parallel activation of machines associated with itself. A transition entering a box represents the parallel activation of the corresponding machines, while a transition exiting a box corresponds to a return;
- fork:** control pseudo-node modelling the activation of new processes. Such activation may be either *synchronous* (the forking process is suspended and waits for the activated processes to terminate) or *asynchronous* (the forking process continues its activity along the newly-activated processes);
- join:** control pseudo-node used to synchronize the termination of concurrently executing processes or to force their termination (*preemptive join*).

The vertices corresponding to stable, meaningful control points are called *nodes*, as opposed to *pseudo-nodes*, which are only transient points. Transitions represent changes in the control state of a machine. A transition is labelled with a name and decorated with a *trigger* (an input event originating from the external environment or from other machines, e.g. the presence of messages on a given channel), a *guard* (a Boolean condition on the current contents of variables and channels) and an *action* (one or more statements on variables and channels). For a transition to be fired its trigger must be fulfilled and its guard satisfied. When a transition fires, its action is executed with possible side-effects. In the following τ denotes the trivial trigger (no external event is required), *True* denotes the trivial guard (always satisfied), and ε denotes the empty action (no side effects).

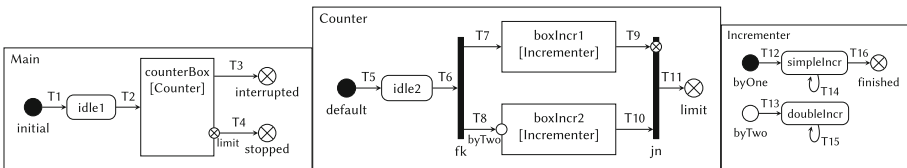


Fig. 1. The *Counting* DSTM specification

Example 1 (The Counting DSTM). Consider the *Counting* DSTM consisting of the machines *Main* (the initial machine), *Counter* and *Incremter* represented

in Fig. 1. Default entering pseudo-nodes are depicted as black circles, entering pseudo-nodes as white circles, final nodes as crossed-out white circles. Boxes are represented by rectangles and decorated with a comma-separated list of associated machines enclosed in square brackets. Nodes are drawn as rounded rectangles and fork and join pseudo-nodes are represented by black bars. Each node and pseudo-node is decorated with its name. Transitions are directed edges from source to target vertices, and are detailed in Table 1, where *Src* and *Trg* are the source and target of the transition, *Dec* is the associated decoration and *Inst* the parameter substitution function. Transitions T1, T5, T12, T13 are *implicit* transitions; T14 and T15, T16 are *internal* transitions; transitions T6 and T11 are, respectively, *entering fork* and *exiting join* transitions; T2 and T7 are *call by default* transitions, while T8 is a *call by entering*. T2, T7 and T8 are transitions with a non-empty substitution function since they enter boxes instantiating parametric machines. T9 and T10 are *return by default*, with the first being a preemptive transition (marked by \otimes). T3, with its non-trivial trigger `signal?`, is a *return by interrupt* while T4 is a well-formed *return by exiting* since its source is `(counterBox, limit)` and `counterBox` instantiates exactly one *Counter* machine and `limit` is an exiting state of such instantiated machine.

A DSTM is *well-formed* if it satisfies a set of syntactical constraints (formally defined in [1]) in order to guarantee that: (a) parameter substitution functions and *call by entering/exiting* transitions are consistent; (b) at each time, the control state of a machine can be located in at most one node; (c) for each *join* pseudonode, there exists a corresponding *fork*. Additionally,

Table 1. Transitions of the *Counting* DSTM

<i>T</i>	<i>Src</i>	<i>Trg</i>	<i>Dec</i>	<i>Inst</i>
T1	initial	idle1	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T2	idle1	counterBox	$\langle \tau, True, \varepsilon \rangle$	$P.to=100$
T3	counterBox	interrupted	$\langle signal?, True, \varepsilon \rangle$	\emptyset
T4	<code>(counterBox, limit)</code>	stopped	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T5	default	idle2	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T6	idle2	fk	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T7	fk	boxIncr1	$\langle \tau, True, \varepsilon \rangle$	$P.limit=P.to$
T8	fk	<code>(boxIncr2, byTwo)</code>	$\langle \tau, True, \varepsilon \rangle$	$P.limit=P.to$
T9	boxIncr1	<code>(jn, \otimes)</code>	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T10	boxIncr2	jn	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T11	jn	limit	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T12	byOne	simpleIncr	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T13	byTwo	doubleIncr	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T14	simpleIncr	simpleIncr	$\langle \tau, x < P.limit, x++ \rangle$	\emptyset
T15	doubleIncr	doubleIncr	$\langle \tau, True, x+=2 \rangle$	\emptyset
T16	simpleIncr	finished	$\langle \tau, x \geq P.limit, \varepsilon \rangle$	\emptyset

Additionally, *exiting fork* and *entering join* transitions can only be labelled with a trivial trigger, guard and action, while boxes instantiated by a fork can only be refined by a single machine.

Example 2. To illustrate the dynamic instantiation capabilities of DSTM and *asynchronous fork* transitions, consider the *Dynamic* DSTM detailed in Fig. 2 and in Table 2. Transition T4 is an *asynchronous fork*, T2 is triggered by the reception of any message on the channel `req` and T3 enters `boxIncr` instantiating an *Incrementer* machine, specified as in Example 1. T6 is an *entering join* transition. Notice that the *Dynamic* DSTM is able to instantiate an unbounded number of concurrent *Incrementer* machines by repeatedly firing transition T2 and

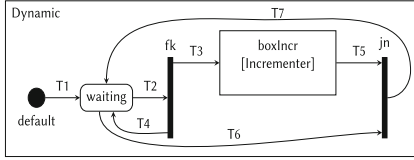


Fig. 2. The *Dynamic* DSTM

Table 2. Transitions of the *Dynamic* DSTM

T	Src	Trg	Dec	$Inst$
T1	default	waiting	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T2	waiting	fk	$\langle req?, True, \varepsilon \rangle$	\emptyset
T3	fk	boxIncr	$\langle \tau, True, \varepsilon \rangle$	P_limit=10
T4	(fk, ↓)	waiting	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T5	boxIncr	jn	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T6	waiting	jn	$\langle \tau, True, \varepsilon \rangle$	\emptyset
T7	jn	waiting	$\langle \tau, True, served++ \rangle$	\emptyset

performing the asynchronous fork T4. Indeed, T4 creates a loop with transition T2, involving the node *waiting* and the fork pseudo-node. When the *Dynamic* machine performs the asynchronous fork T4, it continues its execution in parallel with the activated *Incrementer* machine. Being still active, the process *Dynamic* might fire the transition T2 again, and a second activation of machine *Incrementer* occurs. Hence, this new instance would run in parallel with both the process *Dynamic* and the previously activated instance of *Incrementer*.

The types system in DSTM is based on the one of PROMELA, with the addition of multi-types. Types in DSTM can either be *basic types*, *compound types* or *multi-types*. The *basic types* BT includes the `Int` type for integers, the `Chn` type for channel names and a set of user-defined enumeration types BT_1, \dots, BT_k . *Compound types* are tuples of *basic types*, e.g., the compound type $CT = \langle BT_{j_1}, \dots, BT_{j_k} \rangle$ is a tuple of basic types with $BT_{j_i} \in BT$. *Simple types* contains both *basic types* and *compound types*. A *multi-type* MT is a composition of *simple types*: $MT = \{ST_1, \dots, ST_k\}$. T denotes the set of all types.

Channels allow for communication with the external environment and between internal components via bounded *first-in first-out* buffers. Furthermore, channels are partitioned into the two sets of *internal* and *external* channels. Internal channels, whose names belong to $C_I \subseteq C$, are used for communications between components and are restricted to simple types, whereas external ones, whose names belong to $C_E \subseteq C$, are used for communications with the external environment and are restricted to having bounded buffers of length 1.

2.1 DSTM Semantics

The evolution of a DSTM consists in a sequence of instantaneous reactions called *steps*. A step is a maximal set of transitions that are triggered by the current system state and by the current value of channels. The firing of a transition can have side effects on the available channels and variables. The content sent during a step on an external channel, unlike for internal ones, can only be observed in the next step. DSTM semantics is defined over *ground machines*, namely machines in which actions, triggers and guards contain no parameters (parameters do not hold any value during execution, they serve only as placeholders and

are substituted with actual values before instantiation). Ground machines are obtained from parametric machines by suitably applying parameter-substitution functions. The semantics of transition decorations is defined w.r.t. an evaluation context $\langle \rho, \chi, \eta \rangle$, where ρ associates variables with values, χ evaluates channels content in the current state, while η associates with each external channel its content in the next step. The formal semantics for DSTM is provided by defining a Labelled Transition System (LTS) [1]. The main intuition behind this formalization is that each state s of the LTS model represents a complete configuration (*state*) of the DSTM in a given instant, including the current control locations and the current evaluation context, while a *step* in the DSTM will correspond to a suitably-defined sequence of LTS transitions, each capturing DSTM transition firings. The global control state stores information about the current control state of each active process (ground machine). Since a machine may instantiate multiple machines, the control state can be represented by a tree, called the *control tree*. Each vertex of such a tree is labelled with either a machine, a box or a node. According to the intuition that pseudo-nodes represent only transient non-stable control points, control tree vertices cannot be labelled by pseudonodes. The root of a control tree, labelled by a machine, represents the main (initial) process, having the highest level in the hierarchy. Leaves represent control states in which each currently-active process is in and are labelled by nodes. Internal vertices represent the call hierarchy and cannot be labelled by nodes. Whenever a vertex is labelled by a machine M , it either is the root or is the child of a node labelled by a box instantiating M . If a node is labelled by either a box or a node, then its parent is labelled by the machine the box or the node belongs to.

Definition 1. *The state of a DSTM D is a tuple $\langle CT, Fr, \theta \rangle$ where:*

- CT is a control tree over D , describing the current state of the control flow;
- Fr is the frontier of CT , containing those vertices of CT that can be the source of a transition in the current step;
- $\theta = \langle \rho, \chi, \eta \rangle$ is an evaluation context.

Example 1 (Continued). Consider the *Counting* DSTM depicted in Fig. 1. Some of the *Counting* DSTM's possible control trees are represented in Fig. 3. In the figure, each machine-labelled (resp. box-labelled, node-labelled) vertex is depicted as a diamond \diamond (resp. a square \square , a circle \circ - possibly crossed-out \otimes if labelled by an exiting state). Moreover, each node is decorated with the name of the corresponding machine/box/state.

Tree (a) encodes the control state in which only the *Main* machine is running and is in the *idle1* state. Tree (b) encodes the control state in which the *Main* machine has entered the box *counterBox*, thus instantiating an instance of the *Counter* machine in its state *idle2*. Tree (c) is the control state in which the *Counter* machine, instantiated by *Main* by entering the box *counterBox*, in turn instantiates two distinct instances of the *Incrementer* machine. by entering the boxes *boxIncr1* and *boxIncr2*. The first instance is in the *finished* end state, while the other one is in the *doubleIncr* state.

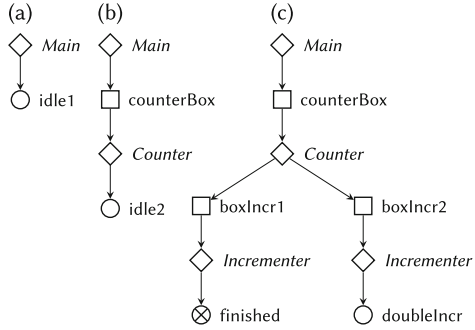


Fig. 3. Control trees of the *Counting* DSTM

DSTM transitions may have source or target in pseudo-nodes which, as said, correspond to transient, unstable control points. Therefore, a transition involving pseudo-nodes may be seen as part of a *super-transition* connecting proper control points. For example, a fork (resp., a join) can be seen as a super-transition connecting one source with multiple targets (resp., multiple sources with one target). *Compound transitions* are able to capture this intuition and allow us to consider only transitions having source(s) and target(s) in proper control points. Hence, there exist three types of *compound transitions*: *simple* (non-implicit transition such that neither its source nor its target are fork or join nodes), *fork* and *join*. The notion of *enabledness* of a transition w.r.t. a DSTM state is as follows. A compound transition of a machine *M* is enabled in a DSTM state *s* if: (a) the guards and triggers of the transition are satisfied in the execution context of *s*; (b) the sources of the compound transition are contained in the frontier of *s*; and (c) no transition of an ancestor of *M* in the hierarchy is enabled. The targets of an executed transition cannot belong to the frontier of the resulting DSTM state, so as to prevent the sequential firing of transitions within the same step. Once the maximality of the current step has been reached (no other transition can be executed in the current step), an implicit *next step* transition occurs. Such transition updates: (a) the frontier with the vertices of the current control tree; and (b) the external channels with new messages, either those produced in the previous step or, if no message was produced for that channel, with non-deterministically generated messages.

Example 2 (Continued). Consider the *Dynamic* DSTM of Fig. 2. Figure 4 shows steps in one of its possible computations. In its initial state s_0 , the DSTM has a control state encoded by tree (S_0) . Suppose that the external environment generates a message on the external *req* channel, thus enabling transition T2.

The compound asynchronous fork $ct_1 = \langle\langle T_2 \rangle, \langle T_3, T_4 \rangle\rangle$ is enabled in the waiting-labelled node. No other compound transitions are enabled, so the first step consists only in ct_1 and in the *next step* transition. When ct_1 fires, the node 1 (labelled by waiting) is replaced by two subtrees obtaining tree (S_1) . Suppose that another message is available on the external channel `req`. Compound transition ct_1 is again enabled in node 1. This time also T14 from the *Incrementer* machine is enabled, and so is the simple compound transition $ct_2 = \langle\langle T_{14} \rangle, \langle T_{14} \rangle\rangle$. The second step consists of two compound transitions ct_1 and ct_2 , which may be executed in any order, followed by the *next step* initialization transition. Execution of step 2 results in the control tree (S_2) , where two instances of the *Incrementer* machine are executing concurrently along with the *Dynamic* machine, which is waiting for new requests in its waiting state.

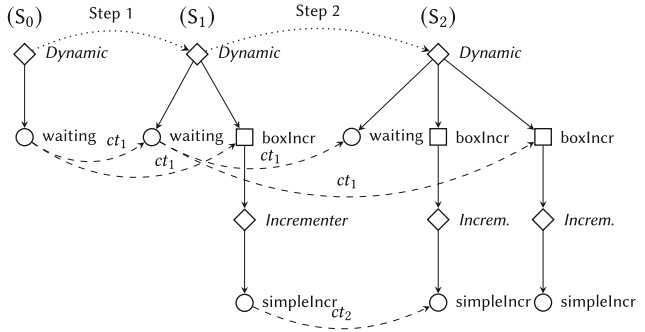


Fig. 4. Steps in a *Dynamic* DSTM computation

3 From DSTM to Promela

Translating a DSTM to PROMELA presents several challenges, due to the substantial differences between the two specification languages. The translation we propose is a two-step process. The first step encodes the vertical hierarchical structure of a DSTM model (boxes) into the PROMELA *proctype* system. The second step transforms the resulting ordinary state machines into an actual PROMELA specification which also takes care of enforcing the step semantics and modelling a possibly non-deterministic environment.

3.1 Encoding the DSTM Vertical Hierarchy

This step transforms each machine of a hierarchical DSTM specification into an ordinary (flat) state machine, by removing all boxes, forks and joins and by substituting them with suitably defined nodes and transitions. Such transitions are also used to model the activation of other flat machines (by means of the PROMELA `run` command) and to ensure a correct handling of machine termination. Each such machine can then be encoded into a PROMELA *proctype*. Note that this transformation does not affect the size of the specification, indeed the size of the resulting model is linear in the size of the original DSTM.

For each machine M a type M_ex is introduced that enumerates all the exiting states of M . Recall that the execution of a PROMELA `run` command associates a

pid to the activated process. The handling of termination is achieved by adding, for each machine M instantiated by a box B , two new channels: a channel of type M_ex , named $chT_B_M_ex$, and a channel of type $\{term, interrupt\}$, called chT_B_M . The first channel is used by the called machine to communicate the reaching of an exiting state to the caller, while the second is used by the caller to issue a termination message to the callee, signalling whether the termination is synchronous or preemptive (i.e., an interrupt).

Each machine activation action has the form `run MachineName(params)`, where `params` is a list containing the following parameters:

- `parent`: the pid of its parent process in the hierarchy;
- `initialState`: the initial state for the instance being instantiated;
- `ch_T_ex` and `ch_T`: the channels required to handle termination.

When removing a box, three different situations may arise, depending on the structure of the DSTM, each dealt with a specific translation schema:

- simple box**: all the transitions entering the box have as source boxes or nodes;
- synchronous fork**: the source of the transition entering the box is a fork pseudo-node and no asynchronous fork transition exiting the fork exists;
- asynchronous fork**: the source of the transition entering the box is a fork pseudo-node and there is an asynchronous fork transition exiting the fork.

Simple Box Schema. In this case the box is substituted by a node having the same name. All transitions whose source (resp. target) is the box are replaced by transitions exiting (resp. entering) the newly-created node, as shown in Fig. 5. The decoration of this transition extends the one of the original transition, in order to model the instantiation of the other machines associated with the box and to handle their termination. As shown in Fig. 5, the triggers and guards of an entering transition are unchanged. A run action is added for each machine instantiated by the box, with the `parent` parameter set to the pid of the calling process.

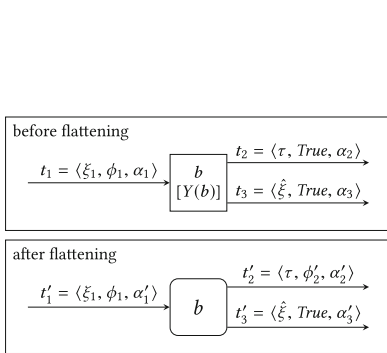


Fig. 5. Simple box flattening

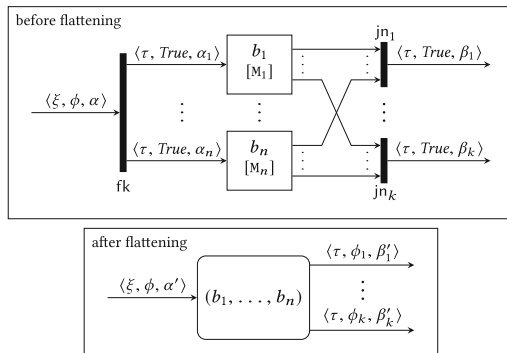


Fig. 6. Synchronous fork flattening

As for the transitions exiting the box, we distinguish the following cases. If the original transition is a *return by exiting*, the guard needs to be enriched with a condition checking for termination of the instantiated machine in the required exit state. Hence, a guard of the form `chT_b_M_ex[?<ex>]` is conjoined with the original guard. If the original transition is a *return by default*, the guard needs to be conjoined with a check for the termination of each of the called machines. To check for a machine termination, regardless of the exiting state, we use a condition of the form `chT_b_M_ex[?<_>]`. If the original transition is a *return by interrupt*, the transition guard need not be enriched. In either case, when a return transition fires, all the called processes must terminate. This is achieved by adding, for each called process, two actions. One of the form `chT_b_M!<msg>`, where `msg` is either `interrupt` or `term`, which sends a termination message to the called process. The second action `chT_b_M_ex?<_>` is used, instead, to clean the corresponding channel used by the terminating machine to signal its termination.

Synchronous Fork Schema. In the synchronous fork case, the calling process suspends itself and waits for the termination of the called processes. In this case, the fork pseudo-node, the boxes called by the fork and the associated (non-preemptive) joins are considered as a single block. The entire block is replaced by a new node and suitably defined transitions to and from that node. The transition modelling the fork operation leads from the source node of the *entering fork* to the newly-introduced node. This transition instantiates the necessary processes by means of appropriate `run` actions, as in the *simple box* case. Each corresponding join operation is modelled by adding a transition from the new node to the target of the original exiting join transition. This transition is decorated with a trivial trigger, a guard requiring the appropriate termination of each machine instantiated by the involved boxes, and an action that takes care of issuing a termination message to each of the instantiated machines and removing messages from the exit-signalling channels.

In the general schema depicted in Fig. 6, the decoration of the transitions modelling the fork are of the form $\langle \xi, \phi, \alpha' \rangle$, where ξ and ϕ are the original trigger and guard of the corresponding *entering fork* transition, and $\alpha' = \alpha \cdot \bar{\alpha}$, with $\bar{\alpha}$ the sequence of `run` actions that activates the processes associated with the called boxes. Each one of the joins jn_i is modelled by a single transition of the form $\langle \tau, \phi_i, \beta'_i \rangle$, where: (i) ϕ_i is the conjunction of the appropriate termination conditions (either by exiting or by default) for each machine instantiated by the fork, as in the case of the *simple box* and (ii) $\beta'_i = \beta_i \cdot \bar{\beta}$, with $\bar{\beta}$ containing the appropriate termination-synchronization actions `chT_B_M!<term>` · `chT_B_M_ex?<_>` for each machine `M` in the box `B` instantiated by the fork.

Asynchronous Fork Schema. After performing an asynchronous fork the calling process continues to run concurrently with the newly instantiated processes. In this case the fork, the boxes entered by the fork and each associated join are considered as a single block and replaced by suitable transitions. The first transition models the fork operation and leads from the source node of the *entering fork* to the target node of the *asynchronous fork* transition, which is a node of the current machine. This transition must also instantiate, by means of appropriate

run actions, the necessary processes that model the called boxes. Note that, in this case the **parent** parameter corresponds to the parent of the calling process, as the new processes being instantiated become siblings of the calling process in the hierarchy tree. In order to model the join operations we add a transition for each join associated with the current fork. Each of these transitions leads from the source node of the *entering join* to the target of the exiting join.

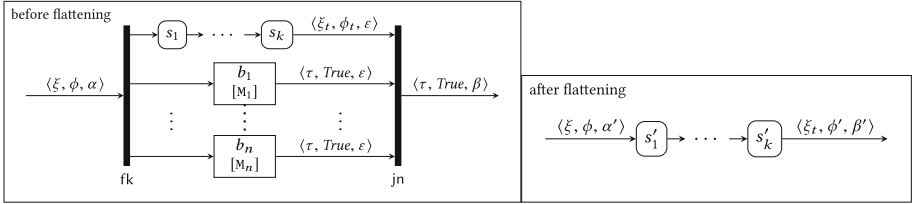


Fig. 7. Asynchronous fork flattening schema

Figure 7 depicts the case of a single fork/join pair, where $s'_i = (s_i, b_1, \dots, b_n)$, with $i \in \{1, \dots, k\}$, to keep track of the concurrently instantiated boxes as well. The decoration of the transition modelling the fork operation is defined exactly as in the case of a synchronous fork. The one modelling the join operation is decorated with $\langle \xi_t, \phi', \beta' \rangle$, where: ξ_t is the trigger associated with the *entering join*; the guard $\phi' = \phi_t \wedge \bar{\phi}$ conjoins the original guard ϕ_t in the entering join with the termination conditions for the instantiated machine; the action $\beta' = \beta \cdot \bar{\beta}$ concatenates the original action of the exiting join with the sequence of termination synchronization actions for the involved boxes.

Handling Preemptive Joins. The fork schemata described above need to be suitably extended in the case the corresponding join is preemptive. In a preemptive join, one or more entering join transitions may be qualified as preemptive. For each such entering join, a distinct transition inheriting the same trigger and guard as the corresponding original preemptive entering join is introduced. Moreover, if the original preemptive entering join is a *return* (either by default or by exiting) with source a box b , the guard is enriched with appropriate conditions requiring the termination of the machine associated with b , as in the previous cases. The action is defined as in the non-preemptive case, the only difference being that termination message issued to the terminating machine is an interrupt and has the form `chT_B.M!<interrupt>`.

3.2 From Flat DSTM to Promela

The PROMELA encoding we propose for a DSTM model is structured as follows:

1. an initial section for global declarations of datatypes, variables, and channels;

2. an **active proctype** named **Engine**, which is the root of the process hierarchy. Its purpose is to start the process modelling the initial machine, manage the proper initialization of external channels before each step, and orchestrate processes in order to simulate the step semantics of DSTM;
3. a **proctype** declaration for each of the n machines M_1, \dots, M_n of the DSTM.

In order to translate a flattened model into a PROMELA specification we need to address the following key points: (1) translation of data-flow elements; (2) orchestration of the concurrent flat machines and correct realization of the steps semantics; (3) encoding of each flat machine into a **proctype**.

Translation of Data-Flow Elements. The mapping of DSTM types and variables to their PROMELA equivalent is rather straightforward, with the DSTM types naturally mapped to the PROMELA types (**mtype** and **datatype**s declared by means of **typedef**). Internal DSTM channels are mapped to PROMELA channels with buffer size equal to the bound of the DSTM channel. If the DSTM channel is a multi-type channel, it is modelled by a set of PROMELA channels, one for each simple type constituting the multi-type. These channels are managed in a way that guarantees that, in each position, at most one of them contains a valid message. This can be achieved by adding a validity **bit** field to each message in the channels. External channels are encoded by channels with buffer size equal to two, with the first position containing the message for the current step and the second position containing the message for the next step possibly produced during the current one. External channels are managed in such a way that the first position in the channel is always filled. This ensures that messages produced in each step are always stored in the second position and that these messages cannot trigger transitions in the current step, as required by the step semantics. To this purpose, an additional validity **bit** field is introduced in every message, so that an empty external channels can be modelled by inserting in the first position a bogus message containing an invalid message.

To comply with the DSTM specification, additional operations on external channels are managed by the **Engine** process. At the beginning of the first step a, possibly-bogus, message for each external channel is non-deterministically generated and placed in the first position. At the beginning of any new step, instead, the messages in the first positions of the external channels, corresponding to the external inputs of the previous step, are removed. For all the channels that remain empty (i.e., no message was generated during the previous step) a, possibly-bogus, message is non-deterministically generated.

Enforcing the Step Semantics. From a global system state $s = \langle CT, Fr, \theta \rangle$, a machine M_i is allowed to execute a compound transition ct if such transition is enabled in state s of the control tree. Due to the encoding of the vertical hierarchy described in the previous section, there are no compound transitions anymore and the above condition can be simplified. An instance of a machine M_i is allowed to execute a transition if:

1. it has never executed during the current step (sequential firing of transitions in the same step is forbidden);

2. none of its descendants in the process hierarchy has executed;
3. none of its ancestors can execute a transition.

In order to simulate the step semantics, we exploit a token-passing mechanism. Each PROMELA process that models an instance of a DSTM machine is required to own a token in order to fire a transition. When a process holding a token is scheduled, it first checks if one of its transitions can be fired. In this case, it performs an enabled transition and consumes the token. If, on the other hand, no transition is executable, the process passes its token to all of its children. A complete top-down propagation of the token in the process hierarchy, starting from the `Engine` process, is called *phase*.

Since a transition fired during a *phase* may enable transitions that were not previously enabled (e.g., by sending messages or modifying the content of shared variables), the token-passing phase needs to be iterated so as to guarantee the maximality of each step. When a phase is concluded without any transition firing, a maximal step is reached.

Recall that sequential firing by the same process and the execution of both an ancestor and a descendant must be avoided during a single step. To this end, during each phase, processes who fire a transition propagate this information upwards in the process hierarchy so as to prevent ancestors from executing transitions (*back-propagation* mode). To implement this mechanism, the following global data structures are used:

- symbolic constants that refer to states of the machines, with an additional `backProp` label;
- a Boolean variable `HasFired`, used to keep track of the fact that at least one transition fired during the last concluded phase;
- an array `HasToken` of `MAX_PROC` bits, used to model token-ownership by active process instances; an array `dyingPid` of `MAX_PROC` bits used to keep track of the `pids` of terminated processes;
- an array `HasExecuted` (resp. `descendantExecuted`) of `MAX_PROC` bits, used to keep track of the fact that a given process (resp. one of its descendants including the process itself) fired a transition during the current step;
- a square matrix `ChildrenMatrix` of bits of size `MAX_PROC`, which encodes the active process hierarchy (`ChildrenMatrix[A].children[B]` is set if the process with pid `B` is a child of the process with pid `A`).

Information about the current state of every machine instance is stored in a `mtype` variable `DSTMstate` local to each PROMELA process. An additional variable `state`, assuming values in `{ready,backProp}`, is used to record whether a given `proctype` is ready to simulate the corresponding machine or is in the back-propagation mode. The step-semantics-enforcing mechanism is detailed as follows:

1. at the beginning of a step, after performing the required management operations for external channels, *Engine* passes the token to the main process and to its siblings. At this point, the global flag `HasFired` and the `Descendant-Executed` flag of every process are set to false;

2. every process owning the token and not having the `DescendantExecuted` flag set, tries to execute a transition. If a transition is executed, the global flag `HasFired` is set to true and a local variable `DSTMstate` is assigned to the machine next state. If no transition is executable, the process passes its token on to its children. In either case, `state` is set to `backProp` and the process enters the back-propagation mode in step 3. If a process is in the back-propagation mode and receives the token, then it is allowed to return to its simulation-ready state, without consuming the token;
3. every process in the back-propagation mode can execute if its `Descendant-Executed` flag is set but its parent flag is not. In this case, the process sets the `DescendantExecuted` flag for its parent as well. When no transition is enabled and the back-propagation is complete (i.e., a deadlock state is reached), the execution moves to step 4;
4. process *Engine* activates and
 - (a) if flag `HasFired` is set, `Engine` starts a new phase. The `hasFired` flag is unset, `Engine` passes the token to its children once again, and the execution continues at step 2;
 - (b) if `HasFired` is unset, then the current phase ended with no transitions fired and the current step is concluded. Execution continues by starting a new semantic step.

This mechanism is implemented in PROMELA by the proctypes schemas reported in Figs. 8 and 9, which are described in detail in the following section.

3.3 Promela Encoding

The complete PROMELA encoding of a DSTM D is as follows. The specification contains n proctypes, one for each machine M of the DSTM. The general schema of such proctypes is reported in Fig. 8. The generic M proctype has the same parameters as the corresponding flat machine, and starts with the declaration of local variables and channels required to handle communication of the termination requirements with its children, if any.

Then, the process enters the main iteration statement (line 4), which terminates in one of the following cases:

- an exiting state of M is reached and a termination request on the channel `chT` is received (line 30);
- it receives an interrupting termination request on the channel `chT` (line 37);
- its parent pid is marked as “dying” in the array `dyingPid` (line 37).

The main iteration statement features an option sequence for each machine state $S \in N \cup En$ guarded by the condition `(state == S && HasToken[_pid]==1) && state==ready`. Each of these option sequences immediately consumes the token (line 8) and, then, enters a selection statement that non-deterministically chooses an enabled transition to execute. This selection construct (line 9) contains an option sequence for each transition t with source state S . Each option sequence is guarded by a condition of the form `(ξ && ϕ && !DescendantExecuted)`, with ξ

```

1 proctype M(pid parent; mtype initial; chan chT; chan chT_ex){
2 // declare channels for termination synchron. with children here
3 byte i; mtype state=ready, DSTMstate=initial;
4 do
5 // for each state  $S \in N_i \cup E_{N_i}$ 
6 :: (DSTMstate==S && HasToken[_pid] && state==ready) ->
7 atomic {
8   HasToken[_pid]=0;
9   if
10 // for each transition  $t$  with  $Src(t) = S, Trg(t) = T, Dec(t) = (\xi, \phi, \alpha)$ 
11 :: ( $\xi$  &&  $\phi$  && !descendantExecuted[_pid]) ->
12    $\alpha$ ; DSTMstate = T; HasFired=1;
13   HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
14 :: else -> // no transition is executable
15   if
16 :: (!HasExecuted[_pid]) -> // did not exec. in this step
17   for (i:0..MAX_PROC-1) { // pass token to children
18     if
19     :: (ChildrenMatrix[_pid].children[i]) ->
20       HasToken[i]=1;
21     :: else -> skip;
22     fi;
23   }
24   :: else -> skip;
25   fi;
26   fi;
27   state = backProp;
28 }
29 // for each exiting state  $ex \in E_{X_i}$ 
30 :: (DSTMstate==ex && chT?[term]) -> {chT?term; goto die}
31 // handle upwards propagation of descendantExecuted
32 :: (state==backProp && descendantExecuted[_pid] &&
33 !descendantExecuted[parent]) ->
34 { descendantExecuted[parent] = 1 }
35 // handle original state restoring after backProp
36 :: (state==backProp && HasToken[_pid]) -> {state=ready}
37 od unless (dyingPid[parent] || chT?[interrupt]) -> {
38   if
39   :: (chT?[_]) -> chT?<<
40   :: else -> skip
41   fi
42   goto die
43 }
44 die: dyingPid[_pid]=1
45 }

```

Fig. 8. Flat machine to proctype

```

1 active proctype Engine() {
2 pid PidMain; byte i;
3 chan chT_Main = [1] of (bit);
4 chan chT_Main_ex = [1] of (bit);
5 PidMain = run Main(_pid, initial, chT_Main, chT_Main_ex);
6 ChildrenMatrix[_pid].children[PidMain]=1;
7
8 nextStep: // starts a new step
9 atomic {
10 // handle external channels management
11 HasFired=0;
12 for (i : 0 .. MAX_PROC-1){
13   HasExecuted[i]=0; descendantExecuted[i]=0;
14   HasToken[i] = ChildrenMatrix[_pid].children[i];
15 }
16 }
17 goto waitTimeout;
18
19 nextPhase: // starts a new phase in the current step
20 atomic {
21 HasFired=0;
22 for (i : 0 .. MAX_PROC - 1){ // give token to children
23   HasToken[i] = ChildrenMatrix[_pid].children[i];
24 }
25 }
26 goto waitTimeout;
27
28 waitTimeout:
29 do
30 :: timeout -> // deadlock
31 if
32 :: (!HasFired) -> goto nextStep;
33 :: (HasFired) -> goto nextPhase;
34 fi;
35 od unless (chT_Main_ex?[_]) -> {chT_Main!<term>}
36 }

```

Fig. 9. The Engine proctype

and ϕ being the trigger and the guard associated with transition t , respectively (line 11). When executed, it performs the actions specified in the transition decoration (line 12). For each run operator occurring within the actions (of the form `run P(X,init,chT,chT_ex)`, where X is either `_pid` or `parent`, in case of asynchronous forks) the pid of the newly-instantiated process is stored in a local variable (`pidTemp = run P(...)`). An assignment statement of the form `ChildrenMatrix[X].children[pidTemp]=1` takes care of updating the process hierarchy accordingly. The token is then given to the newly-instantiated process.

Each option sequence updates the `DSTMstate` variable to the corresponding transition target and sets the flags `HasFired`, `HasExecuted`, and `DescendantExecuted` to *true* (lines 12–13). An additional option sequence, executable only when no transitions are enabled, takes care of passing the token to the process children (line 14–25). After the selection statement, `state` is set to `backProp` (line 27) and the process moves into the back-propagation mode. Lines 30–36 take care of handling process termination, the back-propagation within the current phase, and the restoration of the `state` after the back-propagation. Specifically, for each exiting state, an option sequence of the form shown in line 30

is present. The `DescendantExecuted` back-propagation is handled by the option sequence in lines 32–34. These lines are executable when `state == backProp`, and `DescendantExecuted` is set for the proctype and not for its parent. In this case, the `DescendantExecuted` flag for its parent is set to `true`. The option sequence in line 36 resets the simulation-ready state after the back-propagation.

To conclude the specification, Fig. 9 shows a PROMELA encoding for the process `Engine`. Process `Engine`, after declaring local variables and channels required for handling termination of the initial machine (lines 2–4), starts an instance of the `Main` process (line 5) for that machine and records it as one of its children (line 6). Lines 8–16 are responsible for starting a new semantic step, by executing the statement labelled `nextStep`. This statement manages the initialization of the external channels for the new step, resets the `hasFired` flag and passes the token onto its children (line 14). The `nextPhase` statement (lines 19–25) takes care of starting a new phase, by reinitializing `hasFired` flag and passing the token again to the children. Finally, the `waitTimeout` statement (lines 28–35) forces `Engine` to wait for the current phase to complete. If the phase completes with no transition fired (i.e., `hasFired` is not set), then a new step is initiated, otherwise a new phase starts.

Table 3. Generated PROMELA code statistics

Proctype	Lines of code	Local channels	Options in the main loop
Main (Fig. 1)	164	2	7
Counter (Fig. 1)	140	4	6
Incrementer (Fig. 1)	160	0	7
Dynamic (Fig. 2)	92	2	4
Engine (model in Fig. 1)	57	2	–
Engine (model in Fig. 2)	56	1	–

The application of the transformation rules to the models in Figs. 1 and 2 generates four proctypes for the first model and three proctypes for the second model. Table 3 reports, for each proctype, the number of lines of code, channels and options in the main loop of each process in the PROMELA encoding of the two DSTMs. The prototypical environment for DSTM specification we have implemented is available at <https://github.com/stefanomarrone/dstm>. The repository includes the source code of: (a) a textual editor for DSTM producing DSTM specifications in xml format; (b) the compiler translating a DSTM specification (in xml format) into a PROMELA program (.pml). The textual specifications and the .pml programs for Counter (Fig. 1) and Dynamic (Fig. 2) can be found in the same repository.

Correctness of the Translation. We briefly discuss the correspondence between the DSTM specification and its encoding into PROMELA. In the first phase,

boxes are replaced by machine states, whose labels keep track of the actual boxes they represent. Each DSTM machine is, then, encoded with a single proctype, which records the current state of the machine in the local variable `DSTMstate`. Data types, on the other hand, have a direct correspondence with PROMELA types. According to these observations, each global state of DSTM can easily be mapped into a PROMELA state. Indeed, the hierarchical structure of a DSTM state is encoded in the `ChildrenMatrix` global variable, which connects process instances corresponding to box instances. In other words, all the semantic information encoded in a DSTM state is present in a PROMELA state as well. Moreover, every PROMELA state can be mapped into a DSTM state by abstracting away the additional elements (variables and channels) introduced to simulate the semantics. A DSTM step corresponds to a sequence of PROMELA transitions connecting two PROMELA states in which the control of the process `Engine` is located at the `nextStep` label. With this correspondence in place, it is, then, possible to define a relation between DSTM executions and executions of the PROMELA encoding. Such a relation can be formalized by a *weak bisimulation relation*, where the implementation details, such as the token passing, the back-propagation and the termination mechanisms, are considered non-observable internal actions.

4 Conclusions

In this paper the translation from Dynamic State Machines to PROMELA is presented. DSTM is a concise formalism expressive enough to easily capture peculiar features of multi-process control systems. The automated translation to PROMELA eases the implementation and the integration of tool chains exploiting the usage of formal methods into industrial verification and validation processes. Future work include the study of suitable tunings of the translation in PROMELA in order to mitigate unnecessary state explosion phenomena during the model analysis phase. On the applicative side, we plan to investigate instrumentation methods of the resulting PROMELA code, in particular to support automatic test case generation via model checking with respect to different coverage criteria on the original DSTM model that can take into account the intrinsic hierarchy and modularity of the formalism (e.g., coverage of state/transition of a machine in a specific instantiation context). Finally, the correspondence between executions of a DSTM and of its encoding, as discussed at the end of the previous section, enables model checking of linear time properties with the SPIN engine. To this end, we plan to investigate a suitable extension of LTL, in the same vein of [6], able to contextualize properties within the structure of a DSTM. The extended logic can, then, be translated into classic LTL, by exploiting the correspondence above, and verified with SPIN.

References

1. Benerecetti, M., et al.: Dynamic state machines for modelling railway control systems. *Sci. Comput. Program.* **133**, 116–153 (2017). <https://doi.org/10.1016/j.scico.2016.09.002>
2. Bernardi, S., et al.: Enabling the usage of UML in the verification of railway systems: the DAM-rail approach. *Reliab. Eng. Syst. Saf.* **120**, 112–126 (2013). <https://doi.org/10.1016/j.res.2013.06.032>. <http://www.sciencedirect.com/science/article/pii/S095183201300197X>
3. Bernardi, S., Merseguer, J., Petriu, D.C.: A dependability profile within marte. *Softw. Syst. Model.* **10**(3), 313–336 (2011). <https://doi.org/10.1007/s10270-009-0128-1>
4. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using spin to generate tests from ASM specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) *ASM 2003*. LNCS, vol. 2589, pp. 263–277. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36498-6_15
5. Kölbl, M., Leue, S., Singh, H.: From SysML to model checkers via model transformation. In: Gallardo, M.M., Merino, P. (eds.) *SPIN 2018*. LNCS, vol. 10869, pp. 255–274. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94111-0_15
6. Lanotte, R., Maggiolo-Schettini, A., Peron, A.: Structural model checking for communicating hierarchical machines. In: Fiala, J., Koubek, V., Kratochvíl, J. (eds.) *MFCS 2004*. LNCS, vol. 3153, pp. 525–536. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28629-5_40
7. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing statecharts in promela/spin. In: *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pp. 90–101. IEEE, October 1998. <https://doi.org/10.1109/WIFT.1998.766303>
8. Nardone, R., et al.: Modeling railway control systems in promela. In: Artho, C., Ölveczky, P.C. (eds.) *FTSCS 2015*. CCIS, vol. 596, pp. 121–136. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29510-7_7
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, Cham (2015). https://doi.org/10.1007/978-3-319-17581-2_7
10. Pfügl, H., El-Salloum, C., Kundner, I.: CRYSTAL, critical system engineering acceleration, a truly European dimension. *ARTEMIS Mag.* **14**, 12–15 (2013)
11. Rugina, A.E., Kanoun, K., Kaâniche, M.: The ADAPT tool: from AADL architectural models to stochastic Petri nets through model transformation. In: *2008 Seventh European Dependable Computing Conference*, pp. 85–90. IEEE (2008)