Dynamic state machines for modelling railway control systems

M. Benerecetti [a], R. De Guglielmo [c], U. Gentile [a], S. Marrone [b], N. Mazzocca [a], R. Nardone [a,*], A. Peron [a], L. Velardi [c], V. Vittorini [a]

[a] *Department of Electrical Engineering and Information Technology, University of Naples Federico II, Naples, Italy*
[b] *Department of Mathematics and Physics, Second University of Naples, Naples, Italy*
[c] *Ansaldo STS, Naples, Italy*

A B S T R A C T

Verification and Validation of railway controllers is the most critical and time-consuming phase in a system development life-cycle. It is regulated by international standards, which explicitly recommend the usage of state machines to model the specification of the system under test. Despite the great deal of works addressing the usage of state machines and their extensions, model-based verification and validation processes still lack concise and expressive-enough notations able to easily capture peculiar features of the specific domain of multi-process control systems, on which proper tool chains can be implemented in order to realize effective and automated environments.

This paper introduces a novel class of hierarchical state machines, called Dynamic STate Machines (DSTMs), and proposes an approach for modelling and validating railway control systems, based on the new specification language. Key features of DSTM are recursive execution, parallelism, parameter passing, abortion transition, and communication through global variables and channels, but its main peculiarity resides in the semantics of fork and join operators which allows for dynamic instantiation of machines (processes). The formal semantics of DSTM allows for the definition of verification and validation methodologies supported by automated tools. The paper also describes how DSTM specifications may be mapped to Promela models in order to achieve automated generation of test cases by model checking and Spin.

The work presented in this paper was carried out in the context of an European project and is strongly driven by the industrial necessity of tackling issues concerning the automation of functional system-level testing of modern railway signalling systems. Hence, the language and the proposed approach are illustrated and motivated by applying them to a specific functionality of the Radio Block Centre, the vital core of the ERTMS/ETCS Control System.

Keywords:
Dynamic STate machines
Promela model
ERTMS/ETCS
Control system
Verification and validation

* Corresponding author.
E-mail address: roberto.nardone@unina.it (R. Nardone).

1. Introduction

Verification and Validation (V&V) of critical control systems is partly conducted according to model-based approaches. International standards in the railway domain explicitly recommend the usage of Finite State Machines (for example in the CENELEC norms EN50128 [12] and EN50129 [11]), since the dynamics of critical control systems based on a sequential computation can be abstracted as a state-transition system. An upward trend is to automate the V&V processes by providing toolchains for effective model-based approaches and integrating well-known and assessed tools for quality and architecture management [36,34,3]. As the V&V activities often amount to more than fifty percent of the total development costs, automated solutions are very appealing in industrial settings [33]: they can enhance abstraction and reuse, enable automated verification, improve productivity and reduce the time to market. A key issue is the automated generation of test cases from the system model and its requirement specifications [21,2]. However, we have experienced that, if not properly applied, model-based approaches may introduce complexity rather than simplify industrial processes, and that the available solutions could not fit the safety requirements that dictate a strict control over methods, techniques and tools employed in all phases of the product life-cycle. Essential elements for success are the high-level modelling language which should capture and describe specific features of the system in a natural way, and the tools/techniques supporting the V&V process. In this direction, the main contribution of this paper is the definition of a formal language for modelling critical control systems and a transformational approach enabling the automated generation of test cases.

The work presented in this paper originates within the context of the ongoing ARTEMIS Joint Undertaking project CRYSTAL (CRitical sYSTem engineering AcceLeration) [35]. CRYSTAL is strongly industry-oriented and aims at achieving technical innovation by a user-driven approach, based on the idea to apply engineering methods to real-world industrial *Use Cases* from the automotive, aerospace, rail and health-care sectors.

The modelling language proposed in the paper emerged in an attempt to fit the modelling requirements and the design practises of the industrial partners of the project, specifically of Ansaldo STS (ASTS), within a rigorous and well-founded formal specification and verification approach. ASTS is an international transportation company, leader in the field of railway signalling and integrated transport systems for passenger traffic and freight operation. This work is part of the development of an interoperable testing environment for ERTMS/ETCS control systems [5] addressing one among the project's objectives: provide ready-to-use integrated *tool chains* to reduce V&V and test effort.

In the industrial setting we experienced, one of the main requirements for a modelling language is to be *small*, namely, according to Martin Glinz's insightful remarks [15], "as simple as possible and as rich as needed". It is widely acknowledged that, in order to avoid encoding tricks during the implementation phase, an adequate system specification language should provide a sufficiently rich set of primitive constructs to allow for a natural modelling of the system of interest. Indeed, the larger the gap between the design specification and the actual implementation is, the less useful the results of the design analysis would be. Whenever real industrial systems, as opposed to simple toy examples, need to be designed, the use of complex control flow and data flow constructs is essentially unavoidable. Hence, in order to provide adequate design support tools for industrial-scale safety critical systems, these complex constructs need to be confined within a formal specification framework, equipped with a rigorous and well-defined semantics, so as to make the resulting designs amenable to standard and well-accepted formal analysis processes. In this context, the essential specification primitives must be able to describe concurrent processes, dynamic instantiation and recursive execution of processes, parameter passing, preemptive termination, inter-process communication. In addition, it should allow for the definition of suitable data types in order to model complex data flows, such as different classes of structured messages and their contents.

To meet these requirements, we propose Dynamic STate Machine (DSTM), a formal language explicitly devised for model-based verification and validation of industrial control systems. Although the definition of DSTM is primarily oriented towards high-level modelling of railway control systems and motivated by the modelling issues raised by ASTS, we believe that it is general-enough to model the specifications of control systems in different domains. Indeed, DSTMs are an extension of Hierarchical State Machines, originally proposed in [18]. The key features of DSTM are: a novel semantics for fork and join constructs that allows for the *dynamic and recursive instantiation* of machines, the introduction of *preemptive termination* and the possibility of passing parameters to machines at instantiation time. This paper introduces the formal syntax and semantics of DSTMs and the encoding of DSTM models into Promela, the description language adopted by the Spin model checker [22]. This translation enables both the automated verification of behavioural properties on the modelled system and the automated generation of test cases, by exploiting the model checker's ability to extract counterexamples of violated properties. We apply the proposed approach to the industrial case study proposed by ASTS in CRYSTAL.

Comparison with previous works The work described in this paper is part of a wider research activity, whose primary objective is to contribute to the effective usage of formal methods and notations in industrial processes. In this context, our research investigates the automated generation of formal models from high-level specifications in different application domains (e.g., in [27,8,28,29]). As already mentioned, this paper presents some of the results obtained within the CRYSTAL project. Automated test cases generation is only one step of a more complex workflow, leading to the development of an interoperable testing environment [5]. DSTM was formerly introduced in [31], where a preliminary version of the language is described. With respect to that work, the paper contains a complete and stable definition of DSTM, including its formal semantics, a detailed definition of the control flow and of the data flow (types and variables). The features of DSTM are illustrated by means of a case study proposed by ASTS. In [30], instead, the mapping from DSTM models to Promela was

only outlined, the focus being on the issues concerning the modelling of the non-deterministic environment in Promela. With respect to that work, here a complete semantic-preserving translation from DSTM to Promela is outlined. In particular, we show how the hierarchical structure of a DSTM model is translated into flat state machines, how machine dynamic instantiation, termination and parallelism are translated, and datatypes are mapped into Promela. In addition, we also provide a description of the test cases generation phase and report some experimental results, thereby providing a comprehensive picture of the approach.

Related work A number of formal methods and techniques have been developed by the scientific community in the past decades and applied to the development of critical systems, including railway applications [9]. Though their usage is not largely common in industrial settings, Finite State Machines (FSMs) are widely used for modelling systems where control handling aspects are predominant. Statecharts [18] extend FSMs with hierarchy, concurrency and communication among concurrent components. Hierarchy is achieved by injecting FSMs into states of other FSMs. Concurrency is achieved by composing FSMs in parallel and by letting them run synchronously.

Among the different variants of Statecharts (see [26] for a comparison of their semantics variants), those integrated in UML 2.0 [32] are of widespread use. UML State Machines admit parallel execution by means of composite states and regions. In this formalism, fork (and join) constructs are used to split (and merge) incoming transitions into two or more transitions, terminating on orthogonal target vertexes (i.e., vertexes in different regions of a composite state). Recursive activation and dynamic instantiation are not allowed. Communicating Hierarchical Machines (CHMs) [1] are a variant of Statecharts introduced for succinctness reasons. They introduce the idea of a collection of finite state machines (modules) having nodes and boxes. Each box is associated with one or more instances of modules and a transition entering a box corresponds to a call to the associated modules. In a Statechart there is no notion of module and instance, hence, if multiple instances of the same module are required by the specification, each instance has to be explicitly defined. The introduction of the notion of module also allows for the definition of Recursive State Machines (RSMs), where a module can recursively call itself [1]. Note that Recursive State Machines admit an unbounded number of states, hence they do not belong to the category of Finite State Machines. More recently, a real-time version of this formalism, called Timed Recursive State Machines [6,7], has also been proposed, which allows for the explicit modelling of continuous-time systems. Dynamic Hierarchical Machines (DHMs) have been introduced as an extension of CHMs with dynamic activation of machines [24]: any DHM M_1 can send a DHM M_3 to a concurrent DHM M_2, M_3 starts running either in parallel with M_1 and M_2, or inside M_2, depending on contextual information.

An extension of Extended Finite State Machines (EFSMs) [38] is supported by Modbat [4], a tool to test state-based software systems with potential non-determinism.

Among the commercial specification environments based on Statecharts, we considered STATEMATE [19] and Stateflow. STATEMATE is the first specification environment adopting Statecharts with the original semantics defined for the formalism and revised in [19]. STATEMATE does not allow for fork and join constructs and do not consider dynamic activation of modules. Stateflow is part of the Simulink graphical language used in Matlab. In this formalism, hierarchical state machines can be combined with flow chart diagrams and used to specify the discrete controller in the model of a hybrid system (the continuous dynamics are specified by the capabilities of Simulink). Despite Stateflow being syntactically similar to the Statecharts notation, from the semantic viewpoint [17,16] it prevents any form of non-determinism, and it imposes an explicit strict scheduling in presence of concurrency. Therefore, it essentially corresponds to a graphical notation for sequential imperative languages.

As opposed to Statecharts, the language proposed in this paper adopts the possibility of dynamically instantiate modules in a native manner. The dynamic (possibly recursive) activation of modules is obtained by means of explicit structural constructs of fork and join (and not by message passing as in DHMs). The language here proposed departs from the languages mentioned above mainly by *(i)* allowing for dynamic instantiation of machines and *(ii)* removing the assumptions, implicitly intended in many languages, that control flows exiting from a fork must always be merged by a join operator. DSTM, in fact, allows for recursive activation of the same machine by specifying a novel semantics for fork and join operators. In doing this, we follow the approach adopted in many works. For instance, Requirements State Machines, proposed in [25], enrich the graphical representation of state machines with an explicit specification of the data flow component of the model, in order to facilitate the reviewing process by application experts. The modelling language is inspired by Statecharts at the syntactic level for the control flow part, but the semantics is revised and adapted so as to account for the characteristics of the specific application domain (i.e., avionic systems).

In our case, the syntactical elements are similar to the ones provided by UML 2.0, with the exception of the introduction of the concept of box, the asynchronous characterization of the fork and the additional notion of preemptive join. Some other concepts have been removed, as they can easily be realized in other ways and become redundant: for example, regions inside composite states can be obtained by means of parallel instantiation of machines. The semantics of DSTM, instead, is completely different for what concerns parallel executions. Similarly to Hierarchical Machines, we assume an idea of hierarchy between machines, but enrich this notion with recursive instantiation and parallel execution of machines. This is done by means of the syntactic elements fork and join, borrowed from UML. Hence, the resulting formalism is substantially different, both in syntax and semantics, from Hierarchical State Machines, which do not allow for recursion, from Recursive State Machines, which do not permit parallelism, and from Communicating Hierarchical Machines, which prevent both recursion and dynamic instantiation.

Several works exist which describe concrete applications of model-based testing and model checking in the railway domain [20]. As for the specific application to railway controllers and test case generation, as proposed in this paper, in [10] a test model based on UML/SysML state machines and a black-box test suite are presented for the ETCS Ceiling Speed Monitor. Although this last paper shows that UML/SysML state machines can be associated with a fully formal semantics, neither dynamic instantiation or recursive activation of the same machine is allowed.

Structure of the paper The paper is organized as follows. Starting from the description of the ASTS case study in CRYSTAL, Section 2 illustrates the modelling issues and the requirements for the proposed modelling language. The ASTS case study concerns a functionality of the Radio Block Centre, the vital core of the ERTMS/ETCS control system, which is used as a running example throughout the paper. Section 3 provides the syntax of DSTM while Section 4 defines its formal semantics; the features of the new language are illustrated by using the ASTS case study. Section 5 describes the translation process from DSTM to Promela. The transformations introduced in Section 5 are applied to the DSTM model described in Section 3 and Section 4. The resulting Promela model is used in Section 6 to obtain a complete test suite for the modelled functionality of the Radio Block Centre and the related performance issues are reported. Finally, Section 7 provides some closing remarks and directions for future work.

2. Domain and modelling requirements

In this section, we first describe the requirements for the specification language for modelling control systems as expressed by ASTS and derived from the analysis of the Radio Block Centre, the core component of modern railway signalling systems. The nature of the case study also allows us to illustrate the main motivations for the choices we made in the definition of the new formalism. Section 2.1 provides a high-level description of ERTMS/ETCS and of the Radio Block Centre, Section 2.2 describes the requirements to be addressed by a specification language able to model a multi-process railway control system, Section 2.3 describes the *Communication Management*, a specific functionality of the Radio Block Centre, on which the ASTS *Use Case* in the CRYSTAL project is based. We will use the *Communication Management* procedure as a running example and case study throughout the rest of the paper.

2.1. An ERTMS/ETCS railway control system

The European Rail Traffic Management System/European Train Control System (ERTMS/ETCS) [37] 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 systems with the existing national interlocking systems (IXL). The ERTMS/ETCS specification identifies three functional levels featuring growing complexity. They can be implemented singularly or jointly, and mainly differ in the communication mechanisms adopted in controlling trains. Functional Levels 2 and 3 represent two more cutting-edge solutions than Level 1. Currently, Level 2 is the most widespread choice, according to the deployment statistics.[1] 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 system* is responsible for providing geographical position information to the on-board subsystem; the *track-side system* is in charge of monitoring the movement of the trains.

The CRYSTAL Use Case, provided by ASTS, is the ERTMS/ETCS interoperable testing of the Radio Block Centre (RBC), the most important component of the track-side system. RBC is a computing system, whose aim is to ensure a safe inter-train distance on the track area under its supervision. As shown in Fig. 1, RBC interacts with the on-board system by managing a Communication Session using the EURORADIO protocol and the GSM-R network.

A single RBC is in charge of continuously and concurrently controlling a fixed maximum number of connections with trains, depending on the physical characteristics of the GSM-R network (usually, it manages about 30 trains). The main objective of the train control system is to timely transmit to each train its up-to-date Movement Authority (MA) and the related speed profile. The MA contains information about the distance the train may safely cover, depending on the status of the forward track. RBC is also in charge of managing emergency situations if the communication with one or more trains is compromised. Specifically, when a train is approaching the area supervised by a RBC, it sends a connection request. If the request can be accepted, RBC tries to establish and manage the connection while the train remains under its control. This is performed by dynamically instantiating a new thread for each connection request received by a train; each thread manages the communication with a specific train and performs a certain number of data checks and actions based on the content of the messages received from that train. As a communication can be lost at any time, each thread handles several errors and termination conditions. In particular, if a communication is definitely lost, all pending processes which refer to the delivery of MA to a specific train are terminated and an emergency procedure is started.

2.2. Modelling issues

The high-level description of the RBC control system, given in the previous section, provides the context and the application domain which inspired the modelling formalism proposed in this paper. The ultimate goal is to provide the designer

[1] http://www.ertms.net.

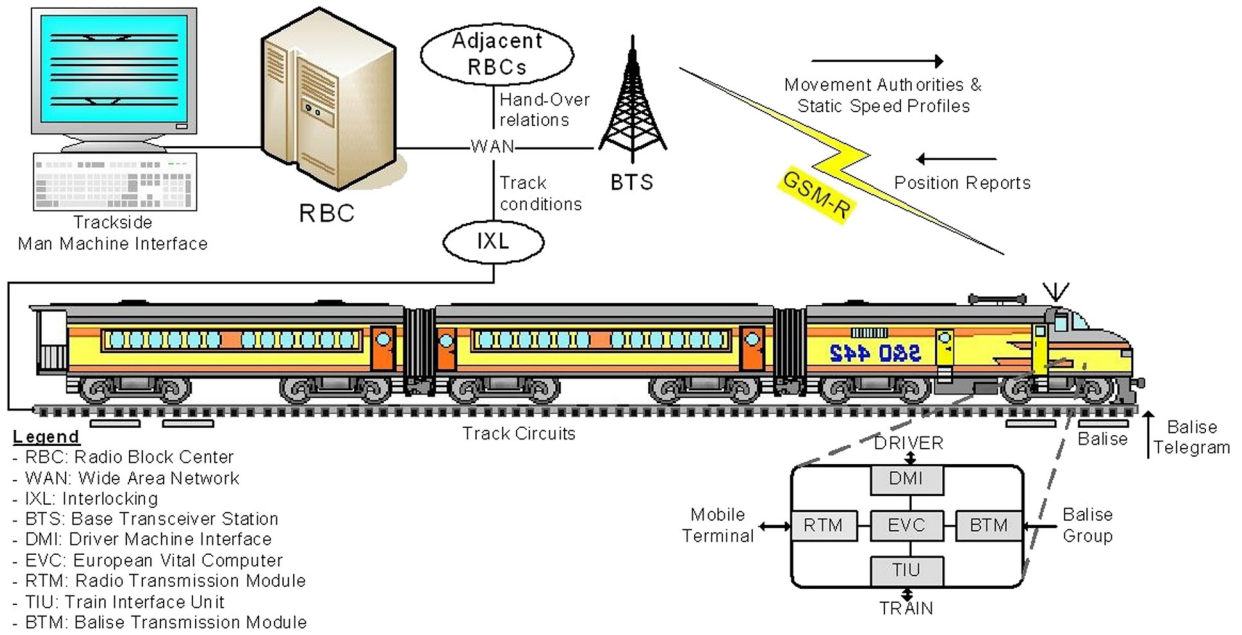

Fig. 1. ERTMS/ETCS Level 2.

with a formal notation sufficiently rich to allow for a natural modelling of control systems of industrial scale. Such a language must provide *primitives* suitable to express specific modelling issues and to avoid intricate encodings of complex behaviours. While the language has been devised in the context of railway control systems, it has a very general flavour that makes it applicable in other industrial contexts.

The description of RBC in Section 2.1 suggests that the language should provide primitives able to naturally describe concurrent threads, dynamic instantiation and recursive execution of processes, parameter passing, preemptive termination and interprocess communication. In addition, it should allow for defining suitable data types in order to model complex data flows, such as ERTMS-compliant protocols, which naturally requires fields-structured messages.

Another important requirement is that such a language should have rigorous and executable semantics, so as to allow for standard and automated V&V activities, such as simulation, formal verification via model checking techniques, and automated test cases generation.

Specifically, the key features that need to be naturally handled by the language are the following.

R1 *Concurrency*: processes may need to run and handle multiple subprocesses, which can execute synchronously (the parent process waits for the termination of the child, similarly to the call/return paradigm) or asynchronously (i.e., concurrently). In addition, the number of subprocesses may be dynamically determined (dynamic instantiation).

R2 *Modularity and parametric processes*: the definition of parametric process schemas that can be instantiated either statically or dynamically, possibly with different parameter values. This feature allows for succinct and reusable specifications, and it is a crucial feature for any specification language oriented to the modelling of large-scale systems.

R3 *Preemptive termination of processes*: error handling is a vital part of the control process, therefore a succinct process abortion mechanism becomes an essential element of the language.

R4 *Interprocess communication*: both point-to-point and broadcast communication mechanisms should be provided as primitives. In particular, communication should not be intended in a pure form, but as conveying structured data either via communication channels or shared variables.

R5 *User-defined data types*: in order to deal with complex data flows, a suitable set of user defined data types should be available (e.g., enumeration typed and flat records).

As a consequence, an approach adopting a modelling language based on standard UML 2.0 [32] state diagrams, as originally proposed in [27], does not seem adequate in the context of railway control systems. In fact, UML state diagrams, on the one hand, lack a standard and well-accepted formal semantics and, on the other, do not allow for some crucial features, such as explicit constructs for modelling dynamic instantiation and parametric instantiation of processes.

2.3. The ASTS case study

The ASTS case study in the CRYSTAL project is the *Communication Management* functionality of RBC, which exhibits all the characteristics described in Section 2.2. It is in charge of establishing and managing the communication session with the

trains. It also handles possible failures, by managing the activation of proper emergency procedures after the occurrence of permanent failures. The *Communication Management* procedure includes three steps, from the entering of a train in the track area controlled by RBC to its leaving. A concise description of these steps, covering all the main issues, is detailed below. The described procedure does not differ significantly from the functionality implemented in a real system, nevertheless it has been slightly modified in respect of any intellectual property rights of ASTS. It also involves a lower number of exchanged messages, but its functional behaviour and the complexity of data are the same of a real-world industrial system. For the sake of exposition, the version of the procedure considered below omits some of the messages and focuses, instead, on the dynamic and parametric instantiation of processes, on the preemptive termination and on the definition of complex data structures.

Step 1: communication establishment When a train needs to establish a safe connection with RBC, it sends a *connection request* message for the initiation of the communication session. RBC may accept only connection requests from a limited number of trains: this number depends on the physical features of the communication radio channel. If the request can be handled, RBC sends to the train a *connection accepted* notification. If the new connection request exceeds the bound, RBC refuses the connection, by sending a *connection refused* message to the train. After the acceptance of a request, this process remains active and waits for other communication requests (Requirements *R1*, *R2*).

Step 2: session establishment Once a connection request is accepted, the communication session between the train and RBC must be established. If this procedure succeeds, RBC authorizes the train to start the *mission* procedure (Start of Mission, SoM) and perform the set of actions required to enter the high-speed area under its control. Finally, RBC sends the *System Version* message to the train and the train answers with an *Ack* and a *Session Established* message. The *Session Established* message is a structured message containing an *area* field used by RBC to distinguish between a train that needs to start its mission *(L0 area)* and a train coming from outside a high-speed area *(L1 area)*. In fact, depending on the case, two different procedures need to be performed: the *Start of Mission* procedure in the former case, and the *Entry* procedure, in the latter case. If a message different from the expected one is received during this protocol, the session establishment procedure aborts and the communication with the train is closed (Requirements *R3*, *R5*).

Step 3: management of train movement RBC periodically sends the Movement Authority (MA) to each train and checks for the reception of commands from the Centralized Traffic Control (CTC), where a human operator may raise alarms requiring the train to stop. In this case an *Unconditional Emergency Stop (UES)* message is sent to the train. When the train successfully ends its trip, RBC performs the *End of Mission (EoM)* procedure (Requirements *R3*, *R4*).

3. Modelling railway control systems in DSTM

In this section we provide the full syntax of the DSTM specification language. Throughout this section, we use the case study introduced above as a running example, so as to exemplify the features of the language and its effectiveness in modelling for real-world applications in a quite natural and structured way. The DSTM formalization is given by separating the elements belonging to the control flow (e.g., vertexes, transitions) and those regarding the data flow modelling (e.g., data types, data structures). We start by giving the syntax for modelling the control flow in a parametric way with respect to the syntactic categories for the dataflow in Section 3.1. The syntax for the specification of the dataflow is, then, presented in Section 3.2.

3.1. Syntax: control flow

A Dynamic STate Machine *(DSTM)* represents a complete specification. Given a set P of parameters, a DSTM consists of a sequence of parametric *Machine*s, $M_1, \ldots, M_n$, communicating via a set X of global *Variable*s and a set C of global *Channel*s. Parameters are parametric names for channels or variables, which are actualized when a machine is instantiated at execution time. Channels and variables allow for different kinds of communication among machines, which will be detailed in Section 3.2 where the dataflow language is defined. Both channels and variables are global in scope, and are declared at the same level of the machines. Neither local channels nor local variables are considered in this version of DSTM.

A (parametric) *Machine* M_i represents a module of the specification exploiting a set of parameters $P_i \subseteq P$. It is a state-transition diagram composed by *vertexes* and *transitions* connecting vertexes. At least two vertexes must be present in a machine. Different kinds of vertexes (with different features and constraints) may occur in the definition of a machine. Possible kinds of vertexes are:

- **node:** basic control state of a machine;
- **entering node:** initial (pseudo)node of a machine (a machine may specify more than one entering node, which represent different entry conditions);
- **initial node:** default entering (pseudo)node of a machine (exactly one for each machine);
- **exit node:** exit (or final) node of a machine (a machine may specify more than one exit node, in order to model different termination conditions);

- **box:** models the activation of one or more processes, instances of the machines associated with the box. A transition entering a box corresponds to the instantiation of the machine(s) associated with that box, and a transition leaving a box corresponds to a return from that machine call;
- **fork:** control (pseudo)node which splits an incoming control flow into more outgoing flows. It allows for instantiating one or more processes, either synchronously (the behaviour of the forking process is suspended waiting for the termination of the activated processes) or asynchronously (the forking process behaves in parallel with the activated processes) with the currently executing processes;
- **join:** control (pseudo)node which merges outgoing control flows from concurrently executing processes. It synchronizes the termination of concurrently executing processes or allows to force their terminations, when a *preemptive* termination is required.

Notice that we distinguish between nodes, which corresponds to stable control points during the execution, and pseudonodes, which represent transient control points. Initial, entering, join and fork nodes are pseudonodes, which only have a syntactic role but no semantic counterpart, as opposed to all the other nodes.

Entering (including initial) and exit nodes define the *interface* of a machine. In particular, the first pseudonode of a machine is either the initial node or an entering node; exit nodes correspond to termination states of a machine, and can be seen as a way to specify different possible return values.

As usual, a transition represents a change of the control state. The firing of a transition is activated when a *trigger* occurs (an input event is issued by environment) and a *guard* on the current content of variables and channels is fulfilled. When a transition fires, an *action* is performed with a possible side effect on the content of variables and channels. In the rest of this section, we shall assume the availability of a set of parameters P, and suitably defined syntactic categories of *triggers* Ξ_P, *guards* Φ_P, and *actions* $\mathcal{A}_P$, all defined over parameters in P. The precise definition of these syntactic categories will be given in Section 3.2. We shall denote the trivial trigger (no external signal required) with τ, the trivial guard with *True* and the empty action with ϵ.

When a transition enters a box, the associated parametric machines need to be instantiated with suitable values for the parameters. A *parameter substitution* function, mapping parameters to actual values, is associated with this kind of transitions. For this reason, in the following definition of DSTM, we shall assume the existence of the syntactic category of the parameter substitution functions over the set of parameters P, denoted by Υ_P. A precise definition of this syntactic category is not essential at this point, and is, therefore, relegated to Section 3.2.

Definition 1 *(Dynamic STate Machine).* A DSTM D is a tuple $\langle M_1, \ldots, M_n, X, C, P\rangle$, where:

- X (resp., C and P) is a finite set of variables (resp., channels and parameters);
- M_1 is the initial machine over X, C (no parameters are allowed in the initial machine);
- M_i, with $i \in \{1, \ldots, n\}$, is a machine over X, C and P of the form

  $\langle P_i, N_i, En_i, df_i, Ex_i, Bx_i, Y_i, Fk_i, Jn_i, \Lambda_i\rangle$, where:
  - $P_i \subseteq P$ is the local set of parameters of machine i (P_1 must be the empty set);
  - N_i is a finite set of nodes;
  - En_i is a finite set of entering pseudonodes;
  - $df_i \in En_i$ is the initial pseudonode (default);
  - $Ex_i \subseteq N_i$ is a set of exit nodes;
  - Bx_i is a finite set of boxes;
  - $Y_i : Bx_i \to \{1, \ldots, n\}^\star$ assigns to every box a sequence (list) of machine indices;
  - Fk_i is a finite set of fork pseudonodes;
  - Jn_i is a finite set of join pseudonodes;
  - $\Lambda_i = \langle T_i, Src_i, Dec_i, Trg_i, Inst_i\rangle$ is the structure defining the set of transitions of M_i, where:
    - T_i is a finite set of transition labels;
    - $Src_i : T_i \to Source_i$ associates a source to each transition label, where $Source_i = (N_i \setminus Ex_i) \cup En_i \cup Bx_i \cup (Bx_i \times Ex(D)) \cup Fk_i \cup (Fk_i \times \{\downarrow\}) \cup Jn_i$ and $Ex(D) = \bigcup_{1\le j\le n} Ex_j$;
    - $Dec_i : T_i \to \Xi_{P_i} \times \Phi_{P_i} \times \mathcal{A}_{P_i}$ associates each transition with its decoration, namely the trigger, guard and action of Λ_i;
    - $Trg_i : T_i \to Target_i$ associates a target to each transition, where $Target_i = N_i \cup Bx_i \cup (Bx_i \times En(D)) \cup Fk_i \cup Jn_i \cup (Jn_i \times \{\otimes\})$ and $En(D) = \bigcup_{1\le j\le n} En_j$;
    - $Inst_i : T_i \rightharpoonup (\Upsilon_P)^*$ is a partial function assigning a sequence of parameter substitution over P to a transition.

The pairing of Fk_i with the symbol $\downarrow$ (resp., of Jn_i with the symbol $\otimes$) is used to qualify a source fork as asynchronous (resp., a destination join as preemptive).

Running example *(part 1).* As an example, let us consider the DSTM:

$D = \langle M_1, M_2, M_3, M_4, \ldots, M_{12}, X, C, P\rangle$, where:

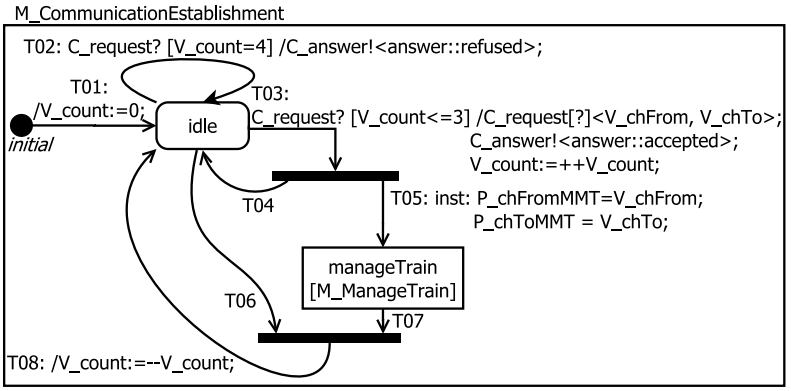

Fig. 2. COMM model: *M_CommunicationEstablishment*.

Table 1
The transition structure Λ_1 of the machine *M_CommunicationEstablishment*.

T_1	Src_1	Trg_1	Dec_1	$Inst_1$
T01	*initial*	*idle*	$\langle \tau, True, V\_count := 0 \rangle$	$\emptyset$
T02	*idle*	*idle*	$\langle C\_request?, V\_count = 4,$ $C\_answer! < answer :: refused > \rangle$	$\emptyset$
T03	*idle*	*fk*	$\langle C\_request?, V\_count \leq 3,$ $C\_request[?] < V\_chFrom, V\_chTo >;$ $C\_answer! < answer :: accepted >;$ $V\_count := ++V\_count \rangle$	$\emptyset$
T04	$(fk, \downarrow)$	*idle*	$\langle \tau, True, \epsilon \rangle$	$\emptyset$
T05	*fk*	*manageTrain*	$\langle \tau, True, \epsilon \rangle$	$P\_chFromMMT = V\_chFrom$ $P\_chToMMT = V\_chTo$
T06	*idle*	*jn*	$\langle \tau, True, \epsilon \rangle$	$\emptyset$
T07	*manageTrain*	*jn*	$\langle \tau, True, \epsilon \rangle$	$\emptyset$
T08	*jn*	*idle*	$\langle \tau, True, V\_count := --V\_count \rangle$	$\emptyset$

M_1 is the machine *M_CommunicationEstablishment* of Fig. 2 and M_2 is machine *M_ManageTrain*. Here we shall focus on the control flow structure of the machine, ignoring the data flow details that will be discussed and explained in the next subsection. Nodes and boxes are depicted as rounded boxes and squares, respectively. The default initial node is depicted as a black bullet. Fork and join pseudonodes are drawn as horizontal bars. As for transition decorations, the first element is a trigger, guard is included in square brackets, action is prefixed by the symbol /, while the symbol ; is used as separator of atomic actions. Finally, parameter substitution functions are prefixed by *inst* :.

The machine is defined as follows:

$$M_1 = \langle P_1, N_1, En_1, df_1, Ex_1, Bx_1, Y_1, Fk_1, Jn_1, \Lambda_1 \rangle, \text{where:}$$

- $P_1 = \emptyset$; $N_1 = \{idle\}$; $En_1 = \{initial\}$; $df_1 = initial$; $Ex_1 = \emptyset$; $Bx_1 = \{manageTrain\}$; $Y_1 = \{(manageTrain, 2)\}$; $Fk_1 = \{fk\}$; $Jn_1 = \{jn\}$;
- $\Lambda_1 = \langle T_1, Src_1, Dec_1, Trg_1, Inst_1 \rangle$ defines the set of transitions of M_1 as reported in Table 1.

Note that the fork pseudonode *fk* is an asynchronous fork since the transition *T04* is marked by $\downarrow$ in Table 1, and that *T05* is the only transition associated with a parameter substitution being the only transition activating a parametric machine (the parameter substitution actualizes two parametric channel names with actual channel names). ■

The transitions allowed in a DSTM belong to predefined classes, reported in Table 2, based on the types of their sources, targets and decorations. The decorations of the transitions belonging to some of those classes may not contain triggers, guards or actions. To this end, we say that a machine M_i has *well-formed transitions* if each transition $t \in T_i$ complies to one of the forms reported in Table 2. For instance, an *implicit* transition of machine M_i starts from an entering node (En_i) of M_i, leads to a node (N_i) of M_i and its decoration must have a trivial trigger τ, a trivial guard *True* and can have some (parametric) action $\alpha \in \mathcal{A}_{P_i}$ (where P_i is the set of parameters of M_i).

Table 2
The syntactic constraints on DSTM transitions ($\xi \in \Xi$, $\hat{\xi} \in \Xi \setminus \{\tau\}$, $\phi \in \Phi_{P_i}$ and $\alpha \in \mathcal{A}_{P_i}$).

Type	*Source*	*Target*	*Decoration*
implicit	En_i	N_i	$\langle \tau, True, \alpha \rangle$
internal	N_i	N_i	$\langle \xi, \phi, \alpha \rangle$
entering fork	N_i	Fk_i	$\langle \xi, \phi, \alpha \rangle$
asynchronous fork	$Fk_i \times \{\downarrow\}$	N_i	$\langle \tau, True, \alpha \rangle$
entering join	N_i	$Jn_i \cup (Jn_i \times \{\otimes\})$	$\langle \xi, \phi, \epsilon \rangle$
exiting join	Jn_i	N_i	$\langle \tau, True, \alpha \rangle$
return by default	Bx_i	$Jn_i \cup (Jn_i \times \{\otimes\})$	$\langle \tau, True, \epsilon \rangle$
	Bx_i	$N_i \cup Bx_i \cup Fk_i$	$\langle \tau, True, \alpha \rangle$
return by exiting	$Bx_i \times Ex(D)$	$Jn_i \cup (Jn_i \times \{\otimes\})$	$\langle \tau, True, \epsilon \rangle$
	$Bx_i \times Ex(D)$	$N_i \cup Bx_i \cup Fk_i$	$\langle \tau, True, \alpha \rangle$
return by interrupt	Bx_i	$Jn_i \times \{\otimes\}$	$\langle \hat{\xi}, True, \epsilon \rangle$
	Bx_i	$N_i \cup Bx_i \cup Fk_i$	$\langle \hat{\xi}, True, \alpha \rangle$
call by entering	$N_i \cup Bx_i$	$Bx_i \times En(D)$	$\langle \xi, \phi, \alpha \rangle$
	$Fk_i \cup Jn_i$	$Bx_i \times En(D)$	$\langle \tau, True, \alpha \rangle$
call by default	$N_i \cup Bx_i$	Bx_i	$\langle \xi, \phi, \alpha \rangle$
	$Fk_i \cup Jn_i$	Bx_i	$\langle \tau, True, \alpha \rangle$

In addition to the general constraints in Table 2, the source of a *return by exiting* transition must be a pair of the form (bx, ex), where $bx \in Bx_i$ is a box of M_i and $ex \in Ex_j$, with $j = \mathrm{Y}_i(bx)$, is an exit node of the machine M_j, associated with bx by the function Y_i. Similarly, for a *call by entering* transition, the target node must be of the form (bx, en), where $bx \in Bx_i$ is a box of M_i and $en \in En_j$ is an entering node of machine M_j, with $j = \mathrm{Y}_i(bx)$. Note that, when a *call by entering* (resp., *return by exiting*) transition is used, the target (resp., source) box must instantiate exactly one machine.

Every transition entering a join pseudonode cannot perform actions (the decoration allows only the empty action). Actions are instead allowed on the transition exiting from the a join pseudonode. Moreover, a *return by interrupt* transition cannot have the trivial trigger τ as trigger.

An empty parameter substitution function is associated with all the transition types except for *call by entering* and *call by default* transitions. If t is a *call by entering* transition, then $Inst_i(t) = \ell$ for some $\ell \in \Upsilon_{P_i}$ defined on all the parameters in P_j. If t is a *call by default* transition and $Y_i(Trg_i(t)) = j_1 \cdots j_s$ is the sequence of indices of (parallel) machines associated with the target box, then $Inst_i(t) = \ell_1 \cdots \ell_s$, for some parameter substitutions $\ell_1, \ldots, \ell_s \in \Upsilon_{P_i}$ and, for all $i \in [1, \ldots, s]$, ℓ_i is defined on all the parameters in P_{j_i} of the called machine M_{j_i}.

Running example *(part 2).* With reference to the machine of Fig. 2, transition *T01* is an *implicit* transition, *T02* is an *internal* transition, *T03* is an *entering fork* transition, *T04* is an *asynchronous fork* transition, *T05* is a *call by default* transition, *T06* is an *entering join* transition, *T07* is a *return by default* transition, and *T08* is an *exiting join* transition.

Observe that the *asynchronous fork* transition *T04* creates a loop with transition *T03*, involving node *idle* and the fork pseudo node, and that the loop does not include any join pseudo node. It is important to note that this form of control flow is not allowed in UML state machines. DSTM overcomes this limitation, thus allowing for dynamic activation of processes. The idea is that when process *M_CommunicationEstablishment* performs the *asynchronous fork T04*, it continues its execution in parallel with the activated process *M_ManageTrain*. Being still active, process *M_CommunicationEstablishment* might fire transition *T03* again, and a second activation of machine *M_ManageTrain* would occur. Hence, this new instance would run in parallel with both process *M_CommunicationEstablishment* and the previously activated instance of *M_ManageTrain* itself.

In the example, the number of activations of machine *M_ManageTrain* that can be concurrently active is at most 4, since the activation process is bounded by a counter (variable *V_count* in the guards of *T02* and *T03*). Different activations of the same machine can also be distinguished by different substitutions of its parameters. In fact, the *call by default* transition *T05* of machine *M_CommunicationEstablishment* instantiates the parametric channel names *P_chFromMMT* and *P_chToMMT* with the two concrete names of channels that are contained in variables *V_chFrom* and *V_chTo* (the details will be clearer after the presentation of the dataflow syntax). The underlying idea in the example is, then, that one can activate an arbitrary number of processes and that each activation can be endowed with a private communication channel.

Conversely, when an active instance of *M_ManageTrain* ends its execution, the *return by default* transition *T07* may fire, by synchronizing with the *entering join* transition *T06*. This results in the deactivation of that instance (the counter of the activations of *M_ManageTrain* is decremented). Notice that, by exploiting this basic mechanism, an unbounded number of activations and deactivation of a machine can be performed. A rigorous semantics of all these features is provided in Section 4.

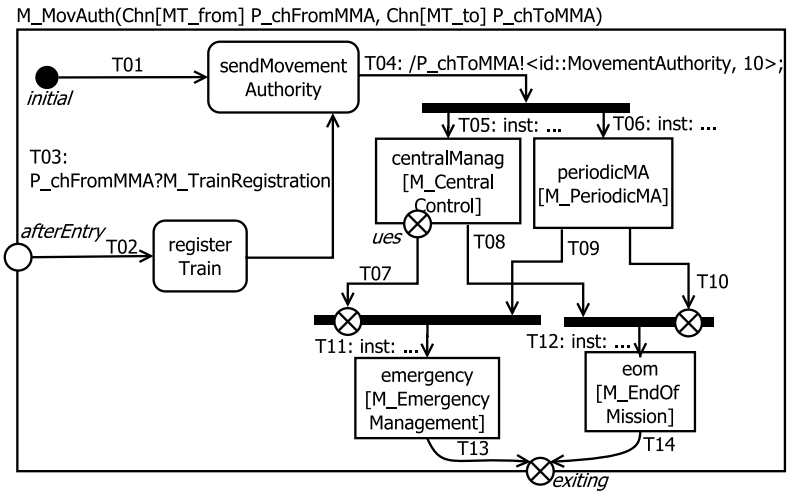

Fig. 3. COMM model: *M_MovAuth*.

Let us now consider machine *M_MovAuth* in Fig. 3. This machine has an input and output interface. This interface corresponds to, beyond the initial node *initial*, the entering node *afterEntry* and the exit node *exiting*, represented as a circle and a crossed circle overlapping the square border of the machine, respectively. Analogously, the exit node *ues* is the terminating state of machine *M_CentralControl*, which is activated within box *centralManag*. The transition *T07* is a *return by exiting* transition, since its source is the couple (*centralManag*, *ues*), where *ues* is the exit node of the machine corresponding to the box *centralManag*. Instead, transitions *T08*, *T09*, *T10* are *return by default* transitions. Moreover, transitions *T07* and *T10* enter the join pseudonodes (jn_1 and jn_2, respectively), whose endpoints are marked with a crossed circle, graphically representing the fact that those transitions perform a preemptive synchronization of the processes involved in their respective join. In particular, when process *M_CentralControl* (resp. *M_PeriodicMA*) terminates at its exit node *ues* (resp., at whatever exit node), transition *T07* (resp., *T10*) fires and the execution of process *M_PeriodicMA* (resp., *M_CentralControl*) is interrupted. As a consequence, termination of the process *M_CentralControl* at its exit node *ues* leads to the termination of the process *M_PeriodicMA* and to the subsequent activation of process *M_EmergencyManagement*. Similarly, termination of process *M_PeriodicMA* leads to the termination of the process *M_CentralControl* and to the activation of process *M_EndOfMission*. ■

Finally, we have to introduce some restrictions in the use of fork and join transitions, in order to guarantee that at each time instant there is at most one node in which the control of a machine can be located. The following well-formedness constraints enforce a correspondence between join pseudonodes and fork pseudonodes, requiring the synchronization of all the processes activated by the corresponding fork transition. The correspondence from join to fork is total (i.e., every join has a corresponding fork) but neither onto nor injective. Indeed, a fork may have no corresponding join (synchronization is not mandatory) or may have more than one corresponding join (multiple synchronization forms). More precisely, a machine M_i is *well-formed* if all its transitions are well formed and the following constraints are satisfied:

1. if a box $bx \in Bx_i$ is the target of either a *call by entering* or a *call by default* transition having as source a fork pseudonode, then
   - there is no other transition entering in *bx*;
   - each transition exiting from *bx* (*return by default*, *return by exiting* or *return by interrupt*) has a join pseudonode as target.
2. for each join pseudonode $jn \in Jn_i$ there is a single corresponding fork pseudonode $fk \in Fk_i$ and
   - if there is an *asynchronous fork* transition exiting from *fk*, then there is an *entering join* transition to pseudonode *jn*;
   - if a box $bx \in Bx_i$ is the target of a call transition (*call by entering* or *call by default*) exiting from *fk*, then either there is no exiting transition from *bx* or there is at least a return transitions (*return by default*, *return by exiting* or *return by interrupt*) from *bx* to *jn*.

The first constraint ensures that a box called by a fork transition cannot be called in any other way and can only have exiting transitions leading to a join. The second constraint enforces the correspondence mentioned above between a join pseudonode and a fork pseudonode. It also requires that a machine *M* performing an *asynchronous fork*, which gives the control back to the *M*, must participate to all the possible joins in correspondence with that fork, so as to ensure that a single control state of *M* results from each join. Moreover, a box *bx* called by a fork either participates with a return transition in all the joins corresponding to that fork, or it never exits.

Running example *(part 3).* With reference to Fig. 2, machine *M_CommunicationEstablishment* is indeed well formed. The first requirement is fulfilled by transition *T05*, which is the only transition having box *manageTrain* as target. The second requirement is fulfilled as well, since the join *jn* is in correspondence with the fork *fk*, and the *asynchronous fork* transition *T04* is in correspondence with *internal* transition *T06* entering the join *jn*.

Let us now consider the machine depicted in Fig. 3. Machine *M_MovAuth* is well formed as well. In fact, both *T05* and *T06* fulfil the first requirement. The second requirement is also fulfilled, since both the joins jn_1 and jn_2 are in correspondence with the fork *fk*, and both jn_1 and jn_2 synchronize the two processes activated by the fork. ■

3.2. Syntax: data flow

Let us now consider the specification of the data flow in a DSTM. We shall start by providing the precise syntax of triggers, guards and actions allowed in the decoration of transitions. We first introduce the type system of the language and the associated domains, then we introduce channels and (global) variables.

Types The set of *basic types* $\mathbf{BT} = \{\mathbf{Int}, BT_1, \ldots, BT_k, \mathbf{Chn}\}$ provides a type **Int** for the integers, a type **Chn** for channels names, and a set of user defined enumeration types $BT_1, \ldots, BT_k$. The domain $\mathcal{D}(\mathbf{Int})$ of integers is $\mathbb{Z}$, the domain $\mathcal{D}(\mathbf{Chn})$ is a set of channel names $C = \{c_1, \ldots, c_h\}$, and the domain $\mathcal{D}(BT_i)$ of enumeration type BT_i is a set of labels $\{l^i_1, \ldots, l^i_{s_i}\}$. We denote with $\mathcal{D}(\mathbf{BT})$ the union of the domains of the basic types. For each basic type we assume that a default value is provided in the corresponding domain. For this purpose we introduce a function $default : \mathbf{BT} \to \mathcal{D}(\mathbf{BT})$.

More complex types can be obtained by composing basic types to form *compound types*. A compound type $\mathbf{CT} = (BT_{j_1}, \cdots, BT_{j_k})$ is defined as a tuple of basic types BT_{j_i}, for $i \in [1, \ldots, k]$. Each element of the domain $\mathcal{D}(\mathbf{CT})$ of **CT** is a tuple of the form $\langle d_1, \ldots, d_k \rangle$, where $d_z \in \mathcal{D}(BT_{j_z})$ for $z \in [1, \ldots, k]$. In other words, the domain $\mathcal{D}(\mathbf{CT})$ is the set of tuples of elements of the basic types composing it. Basic and compound types form the class of *simple types*.

Simple types can be further composed by means of a union operator to form *multi-types*. A multi-type $\mathbf{MT} = \{ST_{j_1}, \cdots, ST_{j_z}\}$ groups together a set of simple types, thus allowing the domain of the resulting type for elements belonging to the domain of any of the simple types ST_{j_i}, with $z \in [1, \ldots, k]$. Hence, the domain of **MT** corresponds to the union of the domains of the composing types. We denote with **T** the set of all types.

Channels To each channel name c in C a *concrete channel* $\widehat{c}$ is associated. The set of concrete channels is denoted by $\widehat{C}$. Channels allow for asynchronous communication both with the environment and among internal components via bounded buffers. Therefore, we have a function $bd : C \to N$ assigning the bound of the buffer associated with any channel name. Each concrete channel has an associated type, either a simple type or a multi-type. Formally, $type : \widehat{C} \to \mathbf{T}$. The type of a concrete channel is actually the type of the message conveyed by the concrete channel. The domain of the contents of a concrete channel $\widehat{c} \in \widehat{C}$ (i.e., the contents of the bounded length buffer associated with the channel) is the set of sequences of length at most $bd(c)$ of elements of its associated type, namely $\mathcal{D}(\widehat{c}) = (\mathcal{D}(type(\widehat{c})))^{\leq bd(c)}$. In order to keep trace of the type of the corresponding concrete channel, the type **Chn** of the channels names can be further decorated with the type of the named channel. Given a channel $\hat{c}$ of type T, we denote with $\mathbf{Chn}[T]$ the decorated type of the name c of $\hat{c}$. We denote with $\mathbf{BT}^+$ the set of *decorated basic type*, containing type **Int**, the enumeration types BT_i and all the decorated types for channel names of the form $\mathbf{Chn}[T]$, for $T \in \mathbf{T}$.

We recall that we distinguish between *internal* and *external* channels. Internal channels are used for communication among internal components and the set of their names is $C_I \subseteq C$. External channels are used to interact with the environment and the set of their names is $C_E \subseteq C$.

The set of internal and external channels are mutually disjoint and form a partition of C ($C = C_I \cup C_E$). In addition, the bound of external channels is restricted to 1, i.e., $bd(c) = 1$, for every $c \in C_E$. Moreover, external channels can be either of a simple type or of a multi-type, whereas internal channels are restricted to simple types.

Variables Let X be the set of (global) variables and P a set of parameters. A typing function $type : X \cup P \to \mathbf{BT}^+$ assigns a decorated basic type to each variable and parameter: variables and parameters of compound types are not allowed. For each variable parameter x (parameter p), the domain of x (p), in symbols $\mathcal{D}(x)$ ($\mathcal{D}(p)$), coincides with the domain $\mathcal{D}(type(x))$ ($\mathcal{D}(type(p))$) of its type.

Running example *(part 4).* Let us consider the machine of Fig. 2. The set of basic types are **Int**, **Chn** and the enumeration type *answer*, whose domain values are {*refused*, *accepted*}. The external channels are *C_answer* and *C_request*. The type of the concrete channel for *C_answer* is the enumeration type *answer* (the channel is used to deliver an answer to an issued request in transitions *T02* and *T03*). The type of the concrete channel for *C_request* is a compound type consisting of a pair of channel type **Chn**. The channel *C_request* is used to receive from the environment a pair of channel names (see transition *T03*) which will be used to instantiate the machine *M_ManageTrain*, activated by transition *T05*.

The machine uses a variable *V_count* of type **Int** which is used to count the number of activated instances of *M_ManageTrain*. Moreover, it has two parameters *P_chFromMMT* and *P_chToMMT* both of type **Chn** which are used to endow the instantiated machine *M_ManageTrain* with a couple of names of channels for input and output communication. ■

We can now introduce the syntax of triggers, guards and actions for DSTM transitions. Preliminary, we need to define the notion of terms, which occur both in the definition of guards and actions.

Terms Terms are freely constructed over variables, parameters, domains of basic types (i.e., basic types literals) and a set Op_1 of unary operators (e.g., $++$, $--$, ...) and a set Op_2 of binary operators (like, e.g., $+$, $-$, $*$, $/$, ...).

More formally, the set of terms over the parameters in P, in symbols Trm_P, is defined as follows

$$trm ::= x \mid p \mid T_i :: l \mid \mathbf{Chn} :: c \mid d \mid len(c) \mid \boxdot_1\, trm \mid trm\ \boxdot_2\ trm$$

where $x \in X$, $p \in P$, $l \in \mathcal{D}(T_i)$, $c \in \mathcal{D}(\mathbf{Chn})$, $d \in \mathcal{D}(\mathbf{Int})$, $\boxdot_1 \in Op_1$ and $\boxdot_2 \in Op_2$. Notice that an enumeration literal is prefixed by its type, and a channel name is prefixed by type **Chn**. The term $len(c)$ denotes the actual length of the buffer associated with channel c.

In order to have admissible term constructions using unary and binary operators (which, for simplicity, are assumed to have only integer operands), we give a type to each term. Terms with undefined type are not admissible. The type $Type(trm)$ of a term trm is defined as follows:

- $Type(x) = type(x)$ and $Type(p) = type(p)$;
- $Type(T_i :: l) = T_i$, $Type(\mathbf{Chn} :: c) = \mathbf{Chn}$, $Type(d) = Type(len(c)) = \mathbf{Int}$;
- $Type(trm_1 \boxdot_2 trm_2) = Type(\boxdot_1 trm_1) = \mathbf{Int}$, if trm_1 and trm_2 are of type **Int**, undefined otherwise.

A term is *well-typed* if its type is defined. In the following we assume that all terms under consideration are well-typed.

Actions We define now the notion of action as a sequence (possible empty) of atomic actions. An atomic action can take one of the following forms: assignment of a term to a variable; the sending of a tuple of values over a channel ($\gamma\,!\,\langle trm_1, \ldots, trm_{k_\gamma}\rangle$); reading (and removing) a tuple of values from a channel and storing them into variables ($\gamma\,?\,\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$); reading a tuple of values from a channel and storing them into variables without altering the content of the channel ($\gamma\,[?]\,\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$). Each element of the tuple $\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$ is either a variable, to which the corresponding element within the message is assigned, or the don't-care symbol _ to skip the corresponding element within the message. The set $\mathcal{A}_P$ of atomic actions over the set of parameters P is, therefore, defined as follows:

$$act ::= x := trm \mid \gamma\,!\,\langle trm_1, \ldots, trm_{k_\gamma}\rangle \mid \gamma\,?\,\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle \mid \gamma\,[?]\,\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$$

where $x \in X$, $type(x) = Type(trm)$, $\gamma \in P \cup C$, $type(\gamma) = \mathbf{Chn}$, $\eta_i \in X \cup \{\_\}$ and if $\gamma \in C$, then $type(\widehat{\gamma}) = (Type(trm_1), \ldots, Type(trm_{k_c}))$. Notice that, since parameters can be assigned to a channel name only at runtime no static type-checking can be defined for these cases. In addition, parameters cannot be assigned a value by an action.

An action is a sequence of atomic actions, with ϵ denoting the empty sequence.

$$\alpha ::= \epsilon \mid a; \alpha \qquad\qquad \text{with } a \in act.$$

Triggers A trigger over P is a boolean expression freely constructed from a set of events (an event is essentially the availability of a message on a channel) by means of standard connectives using parameters in the set P. The definition is as follows:

$$\xi ::= \tau \mid \gamma? \mid \gamma?T \mid \xi \wedge \xi \mid \xi \vee \xi \mid \neg\xi \qquad \text{with } \gamma \in C \cup P$$

where τ is the silent trigger (no event is required for triggering), while a trigger of the form $\gamma?$ (resp., $\gamma?T$) signals the presence of a message (resp., a message of type T) in channel γ. An event of the form $\gamma?$ is sensed when a message is conveyed in channel γ, while the event $\gamma?T$ allows to sense whether a message of a specific type is present in the channel, in particular when the type of the channel is a multi-type. When T is a multi-type, the trigger signals that the channel γ contains a message belonging to one of the types in T.

Guards A guard over a set of parameters P is a boolean expression freely constructed from a set of atomic guards by means of boolean connectives. Let us consider a concrete channel $\widehat{c}$ of (simple) type $(BT_1, \ldots, BT_k)$, atomic guards of the form $c[?\top]$ and $c[?\bot]$ check whether the (buffer of) channel $\widehat{c}$ is full or empty, respectively. Moreover, if $trm_1, \ldots, trm_k \in Trm \cup \{\_\}$ are terms, an atomic guard of the form $c[?\langle trm_1, \ldots, trm_k\rangle]$ checks the content of the message contained in the head of (the buffer of) the channel, namely, for all $i \in [1, \ldots, k]$, either $trm_i = \_$ (the i-th component of the structured message is ignored) or the i-th component of the message at the head of channel $\widehat{c}$ is equal to the value of term trm_i, which must be of type BT_i. In addition, an atomic guard can compare the values of two terms with respect to standard equality and ordering relations (if ordering relations are considered, we assume that terms are of integer type).

The formal syntax of guards is as follows:

$$\phi ::= True \mid \gamma[?\top] \mid \gamma[?\bot] \mid \gamma[?\langle trm_1, \ldots trm_{k_c}\rangle] \mid trm \odot trm \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$$

where $\gamma \in C \cup P$ and $\odot \in \{\geq, \leq, =\}$.

Table 3
Types declared in the specification of the COMM case study.

Basic types		
TypeName	*Domain*	*Description*
bool	{*true*, *false*}	boolean values
answer	{*accepted*, *refused*}	answer to connection requests
area	{*L0*, *L1*}	area field of Session Established messages
version	{*V0*, *V1*, *V2*}	version field of System Version messages
registration	{*registered*, *notRegistered*}	registration field for Train Registration messages
msgId	{*SessionEstablished*, *Ack*, *TrainRegistration*, *SystemVersion*, *MovementAuthority*}	values for message identifiers
Compound types		
TypeName	*Fields Types*	*Description*
M_Request	⟨*Chn*, *Chn*⟩	connection request messages: contains a pair of channel names
M_SessionEstablished	⟨*msgId*, *area*, **Int**, **Int**⟩	Session Established message: contains two identifiers and two integers
M_Ack	⟨*msgId*, *bool*⟩	acknowledge messages: contains an identifier and a boolean value
M_TrainRegistration	⟨*msgId*, *registration*⟩	Train Registration messages: contain an identifier and a registration message
M_SystemVersion	⟨*msgId*, *version*⟩	System Version messages: contains an identifier and a version message
M_MovementAuthority	⟨*msgId*, **Int**⟩	Movement Authority messages: contains an identifier and an integer

Let Ξ_P, Φ_P and $\mathcal{A}_P$ be the syntactic categories of *triggers*, *guards* and *actions*, respectively, over the set P of parameters. An element of Ξ_P (resp. Φ_P, $\mathcal{A}_P$) is a trigger (resp. guard, action) expression.

Terms, guards, triggers and actions in which no parameters occur are called *ground terms*, *ground guards*, *ground triggers*, *ground actions*, respectively.

Parameter substitution A *parameter substitution* function over $\bar{P} \subseteq P$, is a partial function $subst_{\bar{P}} : P \rightharpoonup Trm_{\bar{P}}$ whose domain is the set of parameters P and the range is the set of a terms over $\bar{P}$. Let $\Upsilon_{\bar{P}}$ denote the set of possible parameter substitutions over $\bar{P}$.

Running example *(part 5).* Let us consider the machine of Fig. 2. The transition *T02* has: a trigger *C_request*? requiring the presence of a signal in the external channel *C_request*; a guard $V\_count = 4$, requiring that variable *V_count* be equal to 4; an action *C_answer*! $<$*answer* :: *refused*$>$ which transmits the value *answer* :: *refused* on the external channel *C_answer*. Similarly, the transition *T03* has an action consisting of the sequentialization of three atomic actions: *C_request*[?] $<$ *V_chFrom*, *V_chTo* $>$ reads the content of the external channel *C_request* and assigns the pair of conveyed channel names to the pair of channel variables *V_chFrom* and *V_chTo*; *C_answer*! $<$ *answer* :: *accepted* $>$ transmits the value *answer* :: *accepted* on the external channel *answer*; $V\_count := {+}{+}V\_count$ is a short hand for action $V\_count := V\_count + 1$, which increments variable *V_count*.

The parameter substitution function associated with transition *T05* assigns the terms *V_chFrom* and *V_chTo* (two channel variables) to the parameters *P_chFromMMT* and *P_chToMMT*, respectively. ■

3.3. ASTS case study: the Communication Procedure

Now that the syntax of DSTM has been discussed, we close the presentation of DSTM modelling the COMM Procedure, introduced in Section 2. The complete specification consists of ten machines, the four most relevant of which are completely described here. The machines are depicted in Figs. 2, 3, 4 and 5. We start by describing the data flow specification.

Declaration of types, variables and channels Table 3 reports the definitions and descriptions of the types that need to be declared for the specification of the case study. Two additional multi-types are declared and used to model external channels that can convey messages of different types to or from the trains. The first multi-type *MT_to* is defined as the set of compound types {*M_SystemVersion*, *M_MovementAuthority*}, while the second one *MT_from* is defined as {*M_SessionEstablished*, *M_Ack*, *M_TrainRegistration*}.

Among the declared channels, *C_request* is used by RBC to receive the connection requests from the trains. Each train transmits on *C_request* the names of the two channels used to communicate with RBC. Channel *C_answer* is used by RBC to convey an answer to the train request. In addition, a pair of multi-type channels is used to receive and send messages to the *i*-th train: *C_fromTrain_i*, typed as *MT_from*, and *C_toTrain_i*, typed as *MT_to*.

As already discussed in the running example, the global variables used in the model are the following: the channel variables *V_chFrom* and *V_chTo*, and the integer variable *V_count*.

(Step 1) *Machine M_CommunicationEstablishment (Fig. 2)* The machine already presented in the running examples is in charge of modelling the management of connection requests coming from the trains (Step 1 in the COMM procedure). When receiving a connection request, the machine checks the current number of already connected trains, stored in the variable *V_count*, and if this number is lower then the bound of 4, then it instantiates a new machine *M_ManageTrain* by entering the box *manageTrain*. Variable *V_count* is initialized to zero by the *implicit* transition *T01*, incremented every time a connection request is accepted (transition *T03*) and decremented by transition *T08* every time a machine *M_ManageTrain* is deactivated.

Call by default transition *T05* activates an instance of the machine *M_ManageTrain*, and instantiates its parameters *P_chFromMMT* and *P_chToMMT* by assigning the two channel names that are sent over channel *C_request* when the connection request is issued. Note that the control flow exiting from the fork pseudonode is asynchronous and leads back to the node *idle*. This means that the current instance of machine *M_CommunicationEstablishment* executes in parallel with the newly activated instance of *M_ManageTrain*. Machine *M_CommunicationEstablishment* is, therefore, ready to listen to new train connection requests or to manage a deactivation of the running instances of machine *M_ManageTrain*. When an instance of the machine *M_ManageTrain* terminates its execution, the machine instance is deactivated by joining transitions *T06* and *T07*. The counter variable *V_count* is decremented accordingly. As soon as the number of currently activated instances of *M_ManageTrain* equals the upper bound, any other request is refused (transition *T02*).

(Step 2) *Machine M_ManageTrain (Fig. 4) and Machine M_SessionEstablishment (Fig. 5)* Machine *M_ManageTrain* is instantiated by *M_CommunicationEstablishment* when a train connection request is accepted. This machine models the communication with a specific train (Step 2 in the COMM procedure). Parameters *P_chFromMMT* and *P_chToMMT* are associated (at instantiation time) with concrete names retrieved from the multi-type channels over which the train and RBC communicate. This machine enters node *idle* and, then, instantiates machine *M_SessionEstablishment*, which models the session establishment protocol. Notice that machine *M_SessionEstablishment* has two parametric channels, *P_chFromMSE* and *P_chToMSE*, which are instantiated by transition *T02* with the concrete channel names associated with parameters *P_chFromMMT* and *P_chToMMT*.

The called machine *M_SessionEstablishment* sends the System Version message to the train over the parametric output channel *P_chToMSE*, specifically message *version* :: *V1*. Then, it waits for an acknowledgement message from the train on the parametric input channel *P_chFromMSE* (node *waitForAck*). Notice that channel *P_chFromMSE* is a multi-type channel of type *MT_from* that includes the type of an acknowledgement message *M_Ack*. Therefore, in order to check the presence of an acknowledgement message, the trigger used by transition *T03* is *P_chFromMSE*?*M_Ack*, requiring that the multi-type channel currently contains a message of the specific type *M_Ack*. If the currently delivered message is not of type *M_Ack* (*P_chFromMSE*? && !*P_chFromMSE*?*M_Ack*), the protocol terminates reaching the *abort* exit node (transition *T02*).

After receiving the acknowledgement, the machine waits for the communication of a session establishment message in node *waitForSessEstab*. To check the presence of the communication of session establishment message, transitions *T05* and *T06* use the trigger *P_chFromMSE*?*M_SessionEstablished*, requiring that the corresponding multi-type channel currently contains a message of type *M_SessionEstablished*. Messages of type *M_SessionEstablished* are structured messages with four fields (i.e., {*msgId*, *area*, **Int**, **Int**}). The guards of transitions *T05* and *T06* constrain the value of the second field of the message, requiring the presence of the concrete values *area* :: *L0* and *area* :: *L1*, respectively, but leaving unconstrained the values of the other fields (using the symbol _). (We recall that *area* :: *L0* is a code for a train that starts its mission, whereas *area* :: *L1* is the code from a train coming from a non-high-speed area, and that different protocols are required in the two cases.) Therefore, the different values of the area field lead to different exit nodes (*som* and *entry*, respectively). If the currently delivered message is not of type *M_SessionEstablished*, the protocol terminates reaching the *abort* exit node (transition *T04*).

At this point, if the session establishment protocol terminates in the *abort* exit node, the instance of *M_ManageTrain* terminates its activity as well. Termination of the session establishment protocol in the *entry* or *som* exit nodes leads to the activations of machines *M_Entry* and *M_StartOfMission*, respectively. This allows for specifying the different procedures for trains coming from non-high-speed area and trains starting their mission, respectively.

After termination of the entry procedures (*M_Entry*, for non-high-speed trains, or *M_StartOfMission*, for high-speed ones), *Step 2* of the *Communication Management* procedure is concluded and machine *M_MovAuth* is instantiated in order to give periodically the MA to the train (*Step 3*). Notice that, after the termination of procedure *M_Entry*, machine *M_MovAuth* is activated in its entering node *afterEntry* (by the *call by entering* transition *T06*). Otherwise, it is activated by a *call by default* (transition *T05*), allowing to record the area of the connection establishment.

(Step 3) *Machine M_MovAuth (Fig. 3)* Machine *M_MovAuth* models the activities of *Step 3* in the communication management procedure. As already mentioned, it has two entering modalities. After the activation, it starts two parallel processes by firing fork transition *T04*: an instance of machine *M_CentralControl*, which checks for the reception of commands from the Centralized Traffic Control, and an instance of machine *M_PeriodicMA*, which periodically sends the Moving Authority to the train. The two preemptive joins guarantee that termination of one process forces the termination of the other. For instance, the termination of machine *M_CentralControl* due to a raised alarm, forces the termination of machine *M_PeriodicMA* and

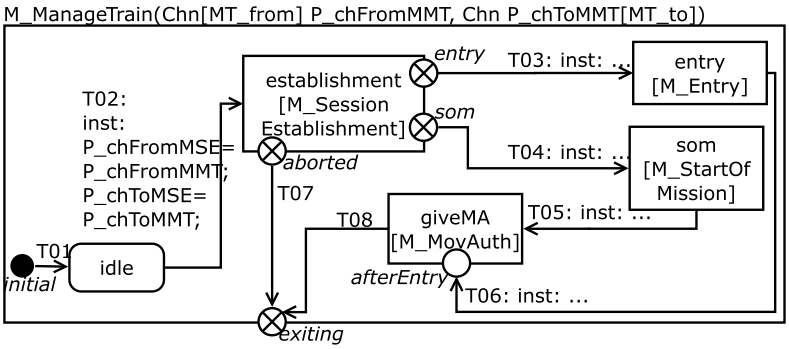

Fig. 4. COMM model: *M_ManageTrain*.

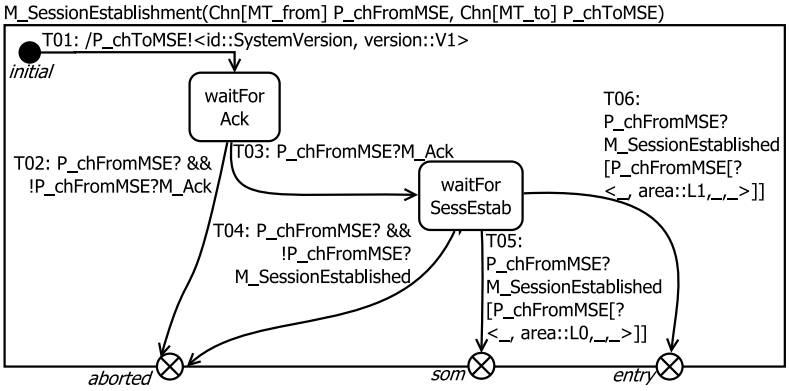

Fig. 5. COMM model: *M_SessionEstablishment*.

activates an instance of machine *M_EmergencyManagement*. If the train successfully ends the trip, machine *M_PeriodicMA* terminates, causing the termination of machine *M_CentralControl* and the activation of an instance of machine *M_EndOfMission*. This last machine successfully terminates the activities of machine *M_MovAuth*. Termination of machine *M_MovAuth* causes, in turn, the termination of its caller process that is an instance of machine *M_ManageTrain*.

4. Semantics of DSTM

The evolution of a DSTM is defined as a sequence of instantaneous reactions, called *steps*. A step is a maximal set of transitions that are triggered by the current set of available events, namely messages conveyed in the external channels. The signals which have been communicated by means of external channels during the firing of a transition cannot trigger other transitions in the same step but only in the next one. For this reason, we distinguish between the content of channels in the current step and the content of external channels generated in the current step and available in the next one.

The central notion of the semantics of a DSTM is the *evaluation context*, relative to which the semantics of terms, actions, triggers and guards is defined. An evaluation context θ is, therefore, a tuple $\langle \rho, \chi, \eta \rangle$, where

- $\rho : X \to \mathcal{D}(\mathbf{BT})$ is the evaluation function for the variables in X;
- $\chi : C_I \cup C_E \to (\mathcal{D}(\mathbf{T}) \cup \{\perp\})^*$ is the valuation function of the internal and external channels for the current step, assigning to each channel a sequence of messages (or $\perp$ for empty locations);
- $\eta : C_E \to \mathcal{D}(\mathbf{T}) \cup \{\perp\}$ is the valuation function for the next step, assigning to each external channel either a message or the symbol $\perp$ when the channel is empty.

We denote with Θ the set of evaluation contexts.

In the rest of this section we shall assume the availability of the following three elements:

- a *denotation function* $[\![trm]\!]_\theta$, which provides the evaluation of a ground term *trm* w.r.t. an evaluation context θ;
- a *satisfaction relation* $\models$, such that $\theta \models \beta$ holds if β, a ground trigger or a ground guard, is satisfied w.r.t. to θ;
- an *action application function* $\theta \xrightarrow{\alpha} \theta'$ that maps an evaluation context θ and a ground action sequence α to a new evaluation context θ', the result from the execution of that action on θ.

The formal definition of these three elements is fairly standard and provided in the appendix. The control flow semantics of DSTM does not crucially depend on how those objects are actually defined, except for the fact that actions are assumed to be *non-blocking*, i.e., any action sequence is always executable in any evaluation context.

DSTM allows for parametric machines, whose parameters are instantiated at execution time when call transitions are performed. As opposed to variables, parameters do not hold values during the execution. They serve as placeholders for the actual values that are dynamically substituted to the parameters when a call operation is executed. The binding of parameters to values is performed by the parameters substitution functions associated with the corresponding call transition. As a consequence, the semantics of a DSTM is defined over *ground machines*, i.e. machines, where terms occurring in actions, triggers and guards do not contain parameters. Ground machines are obtained from parametric ones by applying the appropriate parameter substitutions. Let us first describe how ground machines are obtained from the parametric machines contained in a DSTM. We shall presents the notion of machine instantiation first, then the semantics of a ground DSTM is provided as a Labelled Transition System (LTS) composed of states, corresponding to the possible locations of the control during execution, and transitions, encoding the actual dynamics of the evolution.

Machine instantiation Let P be a set of parameters, ℓ a ground parameter substitution function whose domain of definition is P and the co-domain is the set of ground terms, namely terms where no parameters occur. Let trm be a term such that the set of parameters occurring in it, namely $Parms(trm)$, is contained in P. We denote with $trm[\ell]$ the term obtained from trm by substituting each occurrence of every parameter $p \in Parms(trm)$ in trm with $[\![\ell(p)]\!]_\theta$, the valuation of the term associated with p by the substitution function ℓ. Notice that under the assumptions that trm only contains parameters in P and that ℓ is a ground substitution with domain P, the term $trm[\ell]$ is always ground. Similarly, for an action α (resp., a trigger ξ, a guard ϕ), we define $\alpha[\ell]$ (resp., $\xi[\ell]$, $\phi[\ell]$) as the action (resp., trigger, guard) obtained from α (resp., ξ, ϕ) by substituting each term trm occurring on α (resp., ξ, ϕ) with $trm[\ell]$ and each occurrence of a parameter $p \in P$ in a channel expression of the form $p?\langle\ldots\rangle$, $p[?]\langle\ldots\rangle$, $p[?\langle\ldots\rangle]$ and $p!\langle\ldots\rangle$, with $[\![\ell(p)]\!]_\theta$.

A parameter substitution function associated with a call transition maps parameters into terms, which need not be ground. For instance, in Fig. 4 the parametric machine *M_ManageTrain*, whose parameters are *P_chFromMMT*, *P_chToMMT*, has the call transition *T02* to machine *M_SessionEstablishment*. The parameter instantiation function of this transition requires the substitution parameter *P_chFromMSE* of *M_SessionEstablishment* with *P_chFromMMT*, and parameter *P_chToMSE* with *P_chToMMT*. Clearly, such a substitution is not ground. However, the idea is that when transition *T02* actually fires, the parameters *P_chFromMMT* and *P_chToMMT* have been already replaced by concrete values and the overall substitution will be ground.

To capture this intuition, we need to extend the application of a ground substitution ℓ to another, non-necessarily ground, substitution ℓ'. Let ℓ' be a parameter substitution and $Parms(\ell')$ be the set of parameters occurring in the terms contained in the image of ℓ', i.e., $Parms(\ell') = \{p \mid p \in Parms(\ell'(p'))$, for $p' \in P\}$. Assume, in addition, that $Parms(\ell') \subseteq P$. Then $\ell'[\ell]$ is a (ground) parameter substitution such that for all $p' \in P$, $\ell'[\ell](p') = \ell'(p')[\ell]$, whenever $\ell'(p')$ is defined.

In the example above, let $\ell = \{(P\_chFromMMT, V\_chFrom), (P\_chToMMT, V\_chTo)\}$ be the substitution associated with transition *T05* in Fig. 2, which maps parameters to corresponding variables, and $\ell' = \{(P\_chFromMSE, P\_chFromMMT), (P\_chToMSE, P\_chToMMT)\}$ be the parameter substitution associated with transition *T02* in Fig. 4, of the example above. Then, $Parms(\ell) = \emptyset$, since ℓ is a ground substitution, while $Parms(\ell') = \{P\_chFromMSE, P\_chToMSE\}$. Therefore, we have $\ell'[\ell](P\_chFromMSE) = [\![\ell(P\_chFromMMT)]\!]_\theta = [\![V\_chFrom]\!]_\theta$, as expected. Analogously, $\ell'[\ell](P\_chToMSE) = [\![V\_chTo]\!]_\theta$.

We can now define the ground instantiation $M[\ell]$ of a parametric machine M w.r.t. the parameter substitution ℓ, when the set of parameters occurring in M is $Parms(M) \subseteq P$. Formally, $M[\ell]$ is the machine obtained by substituting for each action α, trigger ξ, condition ϕ and parameter substitution ℓ' of M the corresponding ground object $\alpha[\ell]$, $\xi[\ell]$, $\phi[\ell]$ and $\ell'[\ell]$.

Given a DSTM $D = \langle M_1, \ldots, M_n, X, C, P\rangle$,

$$M(D) = \{M \mid M = M_i[\ell], \text{ with } i \in [1, \ldots, n] \text{ and } \ell \in \Upsilon_{P_i}\}$$

is the set of the possible ground machine instantiations of the (parametric) machines in D.

Control tree and states of a DSTM As usual, formal semantics can be provided by means of a Labelled Transition System (LTS) which is a 4-tuple $L = \langle S, \Sigma, \Delta, S_0\rangle$, where:

- S is a non-empty set of states;
- Σ is a non-empty alphabet of labels;
- Δ is a transition relation, i.e., a subset of $S \times \Sigma \times S$;
- $S_0 \subseteq S$ is a set of initial states.

With reference to a DSTM $D = \langle M_1, \ldots, M_n, X, C, P\rangle$ (see Definition 1), a state $s \in S$ represents the *current* state of D and includes the current control locations and the evaluation context. The following abbreviations are used: $N(D) = \bigcup_1^n N_i$ and $Bx(D) = \bigcup_1^n Bx_i$.

The semantics of a DSTM is given as an LTS containing sequences of a maximal set of transitions (i.e., steps). As we anticipated, the messages generated during a step and sent over an external channel cannot trigger other transitions in

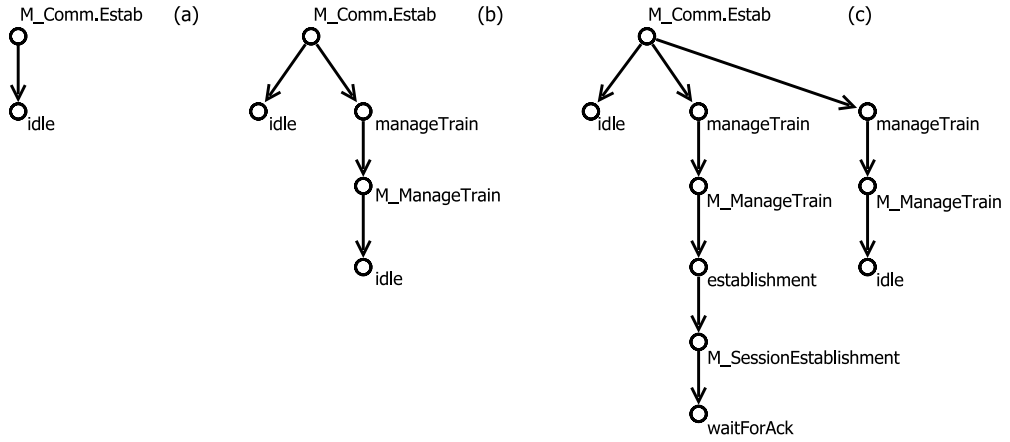

Fig. 6. Control trees (I).

the same step. In addition, a node/box cannot be entered and exited simultaneously in the same step (this is however possible for pseudonodes). As a consequence, if a transition t_i enters into a node n (resp. box b) in a step, and a transition t_j exits from n (resp. b), then t_j cannot fire in the same step. So, sequential firing of transitions is not allowed within a step and only transitions affecting concurrent processes can be performed within the same step. Let us consider first the representation of the current control locations contained in a LTS state. Such a representation describes the currently active processes (instantiated ground machines), the activating processes (calling ground machines) and for each active process the current internal node. Since a process can in general instantiate several concurrent processes by means of call transition or a fork transition, the current control information is represented by a tree, called the control tree, labelled by machines, boxes and nodes.

Running example *(part 6).* Fig. 6 depicts three control trees, representing some possible control trees for machine *M_CommunicationEstablishment* of Fig. 2.

Control tree (a) describes the instance of the machine currently in node *idle*. Control tree (b) results from control tree (a) due to the firing of transition *T03*. The asynchronous fork corresponding to that transition runs the current instance of *M_CommunicationEstablishment* in parallel with an instance of the machine *M_ManageTrain* (instantiated by transition *T05*, which is executed together with *T03*), enveloped in box *manageTrain*. Machine *M_CommunicationEstablishment* is still in node *idle*, since the internal branch of the fork transition (i.e., transition *T04*) leads back to that node. The instantiated machine *M_ManageTrain* is currently in node *idle*. A step further ahead, starting from control tree (b) and firing concurrently transition *T03* of *M_CommunicationEstablishment* (again) and transition *T02* of *M_ManageTrain*, we obtain control tree (c). The firing of call transition *T02* of *M_ManageTrain* activates an instance of machine *M_SessionEstablishment* enveloped in the box *establishment* entering the default node *waitForAck*. The (second) firing of the fork transition *T03* of *M_CommunicationEstablishment* launches (via transition *T05*) a second instance of machine *M_ManageTrain*. Fig. 7 depicts three control trees, representing the control trees obtained by three further steps starting from the control tree (c). In particular, (d) is obtained by two parallel transitions: *T02* in the first activated instance of machine *M_SessionEstablishment* leading to *abort* exit state; *T02* in the second activated instance of machine *M_ManageTrain* leading to the second activation of machine *M_SessionEstablishment* in its default node *waitForAck*. The control tree (e) is obtained again by two parallel transitions: the return by exiting transition *T07* in the first activated instance of machine *M_ManageTrain* which deactivates the first activation of machine *M_SessionEstablishment* and leads to an exit node *exiting*; the internal transition *T03* the second activation of machine *M_SessionEstablishment* leading to node *waitForSessEstab*. Finally, the control tree (f) is obtained by two parallel transitions: the join transition *T08* of machine *M_CommunicationEstablishment*, which deactivates the first activation of machine *M_ManageTrain*, and transition *T06* in the second activation of machine *M_SessionEstablishment*, leading to the exit node *entry*. ■

In general, the root of a control tree represents the highest level process (the instantiation of the starting machine *M_CommunicationEstablishment* in the example above), the leaves represent specific nodes in which each currently active process is waiting, while internal vertexes represent callers and called processes. If a vertex of the tree is labelled by a node or a box of a machine, then its parent is necessarily labelled by that (ground) machine. If a vertex is labelled by a machine, then either it is the root or its parent is necessarily labelled by the box used to instantiate that machine.

Following these ideas we formally define the notion of control tree. A labelled tree $\mathcal{T}$ is a pair $\langle Vx, \lambda \rangle$, where $Vx \subseteq \mathbb{N}^\star$ is a prefix closed set of vertexes (i.e., if $n \in Vx$ and $n' \prec n$ then $n' \in Vx$ with $\prec$ the usual prefix relation between strings) and λ is a function labelling vertexes over a suitable alphabet. The intuition is that each string that encodes a vertex $v \in Vx$ represents the path from the root (represented by the empty string ϵ) to the vertex v itself. For instance, the string n

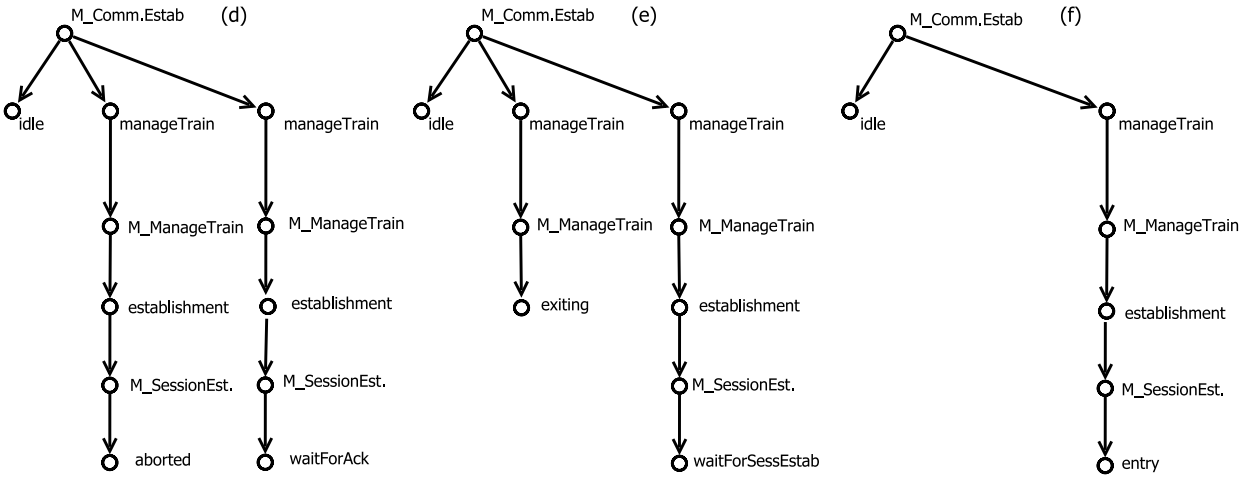

Fig. 7. Control trees (II).

represents the n-th descendant of the root; the string $n.m$ corresponds to the m-th descendant of n, and so on. We denote $Leaves(\mathcal{T})$ the set of leaves of $\mathcal{T}$.

Definition 2 *(ControlTree).* A Control Tree CT over a DSTM D is a pair $\langle Vx_{ct}, \lambda \rangle$, where

- Vx_{ct} is a tree such that each vertex is the parent of at most one leaf;
- λ is a labelling function $\lambda : Vx_{ct} \to M(D) \cup N(D) \cup Bx(D)$ satisfying the following constraints:
  1. $\lambda(\epsilon) = M_1$;
  2. $n \in Leaves(Vx_{ct})$ iff $\lambda(n) \in N(D)$;
  3. if $n = n'.i$ (i.e., n' is the parent of n) with $i \in \mathbb{N}$:
     - $n \in Leaves(Vx_{ct})$ implies $\lambda(n) \in N_j$ and $\lambda(n') = M_j$, for some $j \in \{1, \ldots, n\}$;
     - $n \notin Leaves(Vx_{ct})$ and $\lambda(n) = bx$ with $bx \in Bx_j$ implies $\lambda(n') = M_j$;
     - $n \notin Leaves(Vx_{ct})$ and $\lambda(n) = M_j$ implies $\lambda(n') = bx \in Bx_k$ and j occurs in $Y_k(bx)$, for some $k \in \{1, \ldots, n\}$.

Notice that a vertex of a control tree cannot be labelled by a pseudonode, in accordance with the intuition that control cannot steadily remain in a pseudonode. Moreover, the constraint requiring that each internal vertex be the parent of at most one leaf ensures that a machine cannot be in more than one node at the same time.

A state of a DSTM collects the current control information encoded in a control tree and an evaluation context encoding the current value of variables and channels. Formally:

Definition 3 *(DSTM state).* The *state* of a DTSM is a tuple

$\langle CT, Fr, \theta \rangle$, where:

1. CT is a control tree of the DTSM, which describes the current state of the control flow;
2. Fr, the *frontier* of CT, contains the vertexes of CT that can be source of transitions in the current step;
3. $\theta = \langle \rho, \chi, \eta \rangle$ is an evaluation context.

The frontier Fr within a state is the subset of vertexes of the control tree CT, which can be updated due to a transition firing in the LTS. The frontier is used to avoid the firing of sequences of transitions in the same step. In fact, in order to fire, a transition has to exit only from vertexes belonging to Fr. When a transition of the DSTM fires, the control tree and the evaluation context are updated, while the vertexes corresponding to the sources of the transition are removed from Fr, so as to keep track of the portion of CT that has been already updated.

In the following, for a state $s = \langle CT, Fr, \theta \rangle$ and a guard ϕ (resp., trigger ξ), we shall write $s \models \phi$ (resp., $s \models \xi$), when $\theta \models \phi$ (resp., $\theta \models \xi$). For an implicit transition $t \in T_i$ of a ground machine M_i, with source a pseudonode $pn \in En_i \cup \{df_i\}$ and target $n \in N_i$, we define $explicit_{M_i}(pn) = n$.

The set of initial states S_0 contains states of the form $\langle CT_0, Fr_0, \theta_0 \rangle$, such that:

- $CT_0 = \langle Vx_0, \lambda_0 \rangle$, where $Vx_0 = \{\epsilon, 1\}$, $\lambda_0(\epsilon) = M_1$, $\lambda_0(1) = explicit_{M_1}(df_1)$;
- $Fr_0 = Vx_0$;
- $\theta_0 = \langle \rho_0, \chi_0, \eta_0 \rangle$ is an *initial* evaluation context, where

- ρ_0 gives the initial (default) values of the variables in X, namely $\rho_0(x) = default(type(x))$ for all $x \in X$;
- χ_0 assigns the empty buffer ϵ to each channel in C_I and a (non-deterministically chosen) value, possibly ϵ itself, to each external channel in C_E;
- η_0 assigns $\perp$ to each external channel in C_E.

As an example, the control tree in Fig. 6(a) is the control tree of the initial state of machine *M_CommunicationEstablishment* of Fig. 2.

Semantics of DSTM transitions In order to provide the transitions of the LTS of a DSTM, we have to generalize the notion of DSTM transition. Some DSTM transitions have sources and targets which are pseudonodes. This is the case of transitions (*i*) having source in the entering node of a machine; (*ii*) leading to a fork or join, and (*iii*) exiting from a fork or a join. In particular, a fork can be seen as a hyper transition taking from a source and leading to many targets simultaneously, being the composition of a transition leading to the fork pseudonode and many transitions taking from that fork. Analogously, a join transition can be seen as an hyper transition taking from many sources and leading to a target, being the composition of many transitions leading to the join and one transition taking from the join pseudonode. Therefore, we introduce a notion of *compound transition*. Given a machine M_i a compound transition of M_i is a pair of sequences of transitions $ct = \langle\langle t_1 \ldots, t_k\rangle, \langle t'_1, \ldots, t'_d\rangle\rangle$ where $\{t_1 \ldots, t_k, t'_1, \ldots, t'_d\} \subseteq T_i$. We shall call $t_1 \ldots, t_k$ the *incoming transitions* of ct, and $t'_1, \ldots, t'_d$ the *outgoing transitions* of ct. Besides a trigger $\xi(ct)$ and a guard $\phi(ct)$, to each compound transition two sets of actions are associated: the set $\alpha^{pre}(ct)$ collects the actions of the transitions $\langle t_1 \ldots, t_k\rangle$, while $\alpha^{post}(ct)$ collects those of the outgoing transitions $\langle t'_1, \ldots, t'_d\rangle$.

We distinguish between three different types of compound transitions:

[simple] having the form $ct = \langle\langle t\rangle, \langle t\rangle\rangle$ with t a (non-implicit) transition such that $Src_i(t), Trg_i(t) \notin Jn_i \cup Jn_i \times \{\otimes\} \cup Fk_i \cup Fk_i \times \{\downarrow\}$; $\xi(ct) = \xi(t)$, $\phi(ct) = \phi(t)$, and $\alpha^{pre}(ct) = \emptyset$ and $\alpha^{post}(ct) = \alpha(t)$;

[join] having the form $ct = \langle\langle t_1 \ldots, t_k\rangle, \langle t'_1\rangle\rangle$ such that there exists a join pseudonode $jn \in Jn_i$ and $\{t_1 \ldots, t_k\}$ is the set of (all) transitions having target either jn or $\langle jn, \otimes\rangle$ and t'_1 is the transition such that $Src_i(t'_1) = jn$; we define $\xi(ct) = \bigwedge_{i=1}^{k} \xi(t_i)$, $\phi(ct) = \bigwedge_{i=1}^{k} \phi(t_i)$, $\alpha^{pre}(ct) = \emptyset$, and $\alpha^{post}(ct) = \alpha(t'_1)$. In addition, if there is a j such that the target of t_j is $\langle jn, \otimes\rangle$, then ct is a preemptive join and $\otimes(ct) = Src_i(t_j)$;

[fork] having the form $ct = \langle\langle t_1\rangle, \langle t'_1, \ldots, t'_d\rangle\rangle$ such that there exists a fork pseudonode $fk \in Fk_i$ and $\{t'_1 \ldots, t'_d\}$ is the set of (all) transitions having source either fk or $\langle fk, \downarrow\rangle$ and t_1 is the transition such that $Trg_i(t_1) = fk$; we define $\xi(ct) = \xi(t_1)$, $\phi(ct) = \phi(t_1)$, $\alpha^{pre}(ct) = \{\alpha(t_1)\}$, and $\alpha^{post}(ct) = \bigcup_{1 \le k \le d} \alpha(t'_k)$.

For example, in machine *M_CommunicationEstablishment* of Fig. 2 there is an internal transition $\langle\langle T02\rangle, \langle T02\rangle\rangle$ (implicit transitions are not considered), there is a compound fork $\langle\langle T03\rangle, \langle T04, T05\rangle\rangle$ and a compound join transition $\langle\langle T06, T07\rangle, \langle T08\rangle\rangle$.

Let $s = \langle CT, Fr, \theta\rangle$ and $N \subseteq Fr$ be a set of nodes of (the frontier of) the control tree of s and t a compound transition. The semantics of transition t requires to define: (*i*) when t is enabled in a given state s of the LTS; and (*ii*) the subtrees to add to/remove from the control tree of s to obtain the new state s'. Let us consider in turn these two elements.

Notice, first, that a (proper) DSTM transition t is enabled w.r.t. (the frontier of) a state s only depending on the form of its source node. Given a state $s = \langle CT, Fr, \theta\rangle$ (with $CT = \langle Vx, \lambda\rangle$), a DSTM transition $t \in T_i$ and a vertex $n \in Vx$ of the control tree of s, we introduce a predicate $Enabled(t, n)$ (meaning that transition t is enabled in vertex n of the control tree) and define that $s \models Enabled(t, n)$ iff the subtree of CT rooted in vertex n is contained in the frontier Fr and one of the following conditions, depending on the kind of the transition t, holds:

[internal transition] $Src_i(t) \in N_i$, $n \in Leaves(Vx)$, $\lambda(n) = Src_i(t)$, $s \models \phi(t)$ and $s \models \xi(t)$;
[return by interrupt] $Src_i(t) \in Bx_i$, $\xi(t) \neq \tau$, $\lambda(n) = Src_i(t)$, $s \models \phi(t)$ and $s \models \xi(t)$;
[return by default] $Src_i(t) \in Bx_i$, $\xi(t) = \tau$, $\lambda(n) = Src_i(t)$, $s \models \phi(t)$ and if $n \cdot j \cdot z \in Vx$ then $\lambda(n \cdot j \cdot z) \in Ex$;
[return by exit] $Src_i(t)$ is of the form (bx, ex), with $bx \in Bx_i$, $\lambda(n) = bx$, $n \cdot j \cdot z \in Vx$, for some j and z, with $\lambda(n \cdot j \cdot z) = ex$, $s \models \phi(t)$ and $s \models \xi(t)$.

Notice that the condition requiring that the subtree rooted in n is contained in the frontier verifies that the source of the transition has not been generated during the current execution step and was present in the control tree already at the beginning of the step. This condition prevents the sequential firing of transitions within the same step. Moreover, in the case of an internal transition the source must be mapped onto a leaf n of the control tree. In the case of a return by interrupt, the source is mapped onto a vertex n labelled by a box and no condition is required on the subtree rooted in n. In a return by default, the source is mapped onto a vertex n labelled by a box and each path departing from vertex n has length 2 and ends in an vertex labelled by an exit node (all the processes inside the box have terminated their activities).

In a return by exit, the source is mapped onto a vertex n labelled by a box and there is a path departing from n of length 2 and ending in a vertex labelled by the required exit node (the process inside the box has terminated its activities).

For instance, consider a state s whose control tree is the one depicted in Fig. 7(e). Assume that vertexes of the tree are $\{\epsilon, 1, 2, 3, 2\cdot 1, 2\cdot 1\cdot 1, 3\cdot 1, 3\cdot 1\cdot 1, 3\cdot 1\cdot 1\cdot 1, \cdot 1\}$. Assume, further, that the guard and trigger of transition *T03* are satisfied by s (i.e., the value of variable *V_count* is less than 4 and a message is currently in channel *C_request*). Then, if the subtree containing vertex 1 is in the frontier of s, then $s \models \mathit{Enabled}(T03, 1)$. Similarly, if the subtree rooted at vertex 2 is also contained in the frontier of s, then $s \models \mathit{Enabled}(T05, 2)$ as well (a return by exiting transitions). If, however, state s results from the execution of transition *T07* of machine *M_ManageTrain* applied to vertex 2 in the current step, then the subtree rooted at vertex 2 would not be in the frontier of s, as we shall see later in this section. In this last case, we would have that $s \not\models \mathit{Enabled}(T07, 2)$.

Given a state $s = \langle CT, Fr, \theta\rangle$, a set of vertexes $N = \{n\cdot i_1, \ldots, n\cdot i_k\}$ of Vx (k distinct children of vertex n) and a compound join transition $ct = \langle\langle t_1, \ldots, t_k\rangle, \langle t'_1, \ldots, t'_d\rangle\rangle$ of M_i, we lift the notion of enabledness (as introduced for proper transitions) to compound transitions as follows. Formally,

$s \models \mathit{Enabled}(ct, N)$ holds iff

1. $\lambda(n) = M_i$ and:
2. for all $\bar{n} \in N$ and compound transition ct', if $N' \subseteq Vx$, $n' \in N'$ and $n' \prec \bar{n}$ (and $n' \neq \bar{n}$), then $s \not\models \mathit{Enabled}(ct', N')$ (no proper ancestor of a node in N can be involved in an enabled compound transition);
3. if ct is not a preemptive join, then, for all $j \in \{1, \ldots, k\}$, $s \models \mathit{Enabled}(t_j, n\cdot i_j)$;
4. if ct is a preemptive join then, for all $j \in \{1, \ldots, k\}$:
   - if $t_j = \otimes(ct)$, then $s \models \mathit{Enabled}(t_j, n\cdot i_j)$, and
   - if $\mathit{Src}(t_j) = (bx, ex)$, then $\lambda(n\cdot i_j) = bx$, otherwise $\lambda(n\cdot i_j) = \mathit{Src}(t_j)$.

The first condition requires that the parent n of the vertexes in N belongs to the same machine M_i of the compound transition ct. The second condition ensures that none of the ancestors of the vertexes in N is involved in a currently enabled compound transition (in particular, no ancestors can be interrupted in the current step either by an interrupt transition or by a preemptive join). This condition is crucial as it guarantees that an interrupting transition, enabled in some internal nodes of the control tree, has a higher priority over any other transition enabled in lower subtrees. The third condition requires that when ct is not a preemptive join compound transition, each "source" proper transition t_j of ct be enabled in the corresponding vertex $n\cdot i_j \in N$. Finally, the last condition for a preemptive join compound transition ct ensures that its source preemptive proper transition (namely, $\otimes(ct)$) is enabled, while for each of the remaining source proper transitions, say t_j, it suffices that the corresponding source vertex $n\cdot i_j \in N$ be (labelled with) the source node of t_j.

Consider again the example above, where s is a state whose control tree (and frontier) is that of Fig. 7(e). The compound transition $ct = \langle\langle T06, T07\rangle\langle T08\rangle\rangle$ of machine *M_CommunicationEstablishment* is enabled in state s at vertexes 1 (labelled with node *idle*) and 2 (labelled with box *manageTrain*), i.e., $s \models \mathit{Enabled}(ct, \{1, 2\})$. Indeed, no other compound transition is enabled in the only ancestor ϵ and both the incoming transitions *T06* and *T07* of ct are enabled in s: $s \models \mathit{Enabled}(T06, 1)$ and $s \models \mathit{Enabled}(T07, 2)$.

The effects of a compound transition correspond to the changes of the structure of the control tree and of the valuations of the state in which the transition is taken. With each such transition we, therefore, associate a sequence of trees that will be added to the current control tree, so as to account for the changes in the configuration of the control flow. Given a (compound) transition $ct = \langle\langle t_1, \ldots, t_k\rangle, \langle t'_1, \ldots, t'_d\rangle\rangle$ of the machine M_i, we define sequence of trees $\mathit{Trees}(ct, s) = \langle CT_1, \ldots, CT_d\rangle$ as follows, depending on the target states of the outgoing transitions $\langle t'_1, \ldots, t'_d\rangle$ of ct. For each t'_j, with $j \in \{1, \ldots, d\}$:

- if $\mathit{Trg}_i(t'_j) = n \in N$, then the control tree $CT_j = \langle Vx_j, \lambda_j\rangle$, where $Vx_j = \{\epsilon\}$ and $\lambda_j(\epsilon) = n$;
- if $\mathit{Trg}_i(t'_j) = (bx, en)$, then the control tree $CT_j = \langle Vx_j, \lambda_j\rangle$, where:
  - $Vx_j = \{\epsilon, 1, 1\cdot 1\}$ and,
  - $\lambda(\epsilon) = bx$, $\lambda(1) = M_h[\ell]$, with $\ell = \mathit{Inst}(t'_j)$, and $\lambda(1\cdot 1) = n_h$, where $h = Y_i(bx)$ and $n_h = \mathit{explicit}_{M_h}(en)$;
- if $\mathit{Trg}_i(t'_j) = bx \in Bx_i$, then the control tree $CT_j = \langle Vx_j, \lambda_j\rangle$, where:
  - $Vx_j = \{\epsilon\} \cup \{z \mid 1 \le z \le |Y_i(bx)|\} \cup \{z\cdot 1 \mid 1 \le z \le |Y_i(bx)|\}$ and,
  - $\lambda(\epsilon) = bx$, for each $1 \le z \le |Y_i(bx)|$, $\lambda(z) = M_h[\ell_z]$, with $\ell_z = (\mathit{Inst}(t'_j))_z$ (the z-th parameter substitution function of t'_j), and $\lambda(z\cdot 1) = n_h$, where h occurs in $Y_i(bx)$ at position z and $n_h = \mathit{explicit}_{M_h}(df_h)$.

Running example *(part 7).* Fig. 8 depicts the control trees in $\mathit{Trees}(ct_1, s)$ and $\mathit{Trees}(ct_2, s)$, for the compound transitions $ct_1 = \langle\langle T03\rangle, \langle T04, T05\rangle\rangle$ (see Fig. 2) and $ct_2 = \langle\langle T02\rangle, \langle T02\rangle\rangle$ (see Fig. 4), and a state s, where we assume both transitions are enabled having, for instance, the control tree (b) in Fig. 6.

According to the definition above, the two control trees (see Fig. 8(a)) generated by the execution of ct_1 are due to its outgoing transitions *T04* and *T05*. Since *T04* leads to a node, the first case applies and a tree containing a single vertex is generated. Transition *T05* is a call by default transition to box *manageTain*, which corresponds to machine *M_ManageTrain*.

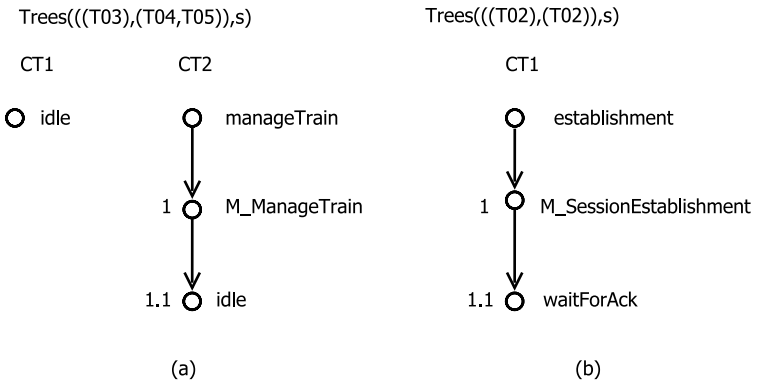

Fig. 8. The trees resulting form the firing the compound transitions $\langle\langle T03\rangle, \langle T04, T05\rangle\rangle$ in Fig. 2 and $\langle\langle T02\rangle, \langle T02\rangle\rangle$ in Fig. 4.

Therefore, the third case applies and a tree is generated where the root ϵ is labelled by *manageTrain*, its only child 1 is labelled by *M_ManageTrain* and the leaf $1 \cdot 1$ is labelled by $explicit_{M\_ManageTrain}(df) = idle$.

Similarly, Fig. 8(b) contains the single tree generated by the firing of compound transition ct_2. This is a simple transition containing a call by default to box *establishment* associated with machine *M_SessionEstablishment*. Once again, the third case above applies. This time the root of the tree ϵ is labelled by *establishment* and its child 1 is labelled by *M_SessionEstablishment*. Moreover, we have a node $1 \cdot 1$, labelled by $explicit_{M\_SessionEstablishment}(df) = waitForAck$. ■

In order to account for the changes in the control tree induced by the firing of a compound transition, we introduce suitable tree transformation operations. Given two trees $\mathcal{T} = \langle Vx, \lambda\rangle$ and $\mathcal{T}' = \langle Vx', \lambda'\rangle$, a node $n \in Vx$ and $i \in \mathbb{N}$, the first operation, $\mathcal{T} \bigcirc_{n \cdot i} \mathcal{T}'$, produces the tree $\mathcal{T}'' = \langle Vx'', \lambda''\rangle$ obtained by substituting the i-th subtree of n in $\mathcal{T}$ (notice that such a subtree may be empty) with the tree $\mathcal{T}'$. Formally, $\mathcal{T}'' = \mathcal{T} \bigcirc_{n \cdot i} \mathcal{T}'$ iff, for every node $\bar{n} \in Vx''$, one of following conditions holds:

- either $\bar{n} \in Vx$, $n \cdot i \not\prec \bar{n}$, and $\lambda''(\bar{n}) = \lambda(\bar{n})$;
- or $\bar{n} = n \cdot i \cdot n'$, for some $n' \in Vx'$, and $\lambda''(\bar{n}) = \lambda'(n')$.

The two main operators, *Remove* and *Update*, are used to delete subtrees consumed by the firing of a transition and to attach the new resulting subtrees. The operator $Remove(CT, N)$, for a set of vertexes N of CT, removes from CT all the subtrees rooted in the vertexes in N, while $Update(C', n, \langle CT_1, \ldots, CT_k\rangle))$, for a vertex n of CT, replaces k empty subtrees of n in CT with the k trees occurring in the sequence $\langle CT_1, \ldots, CT_k\rangle$. We define $Remove(CT, N)$ as the control tree $CT' = \langle Vx', \lambda'\rangle$ where all the nodes of Vx contained in the subtree rooted in some node of N have been removed, while preserving the same labelling as CT on the remaining nodes. Formally, CT' must satisfy the following two conditions:

- $n \in Vx'$ iff $n \in Vx$ and $n' \not\preceq n$, for all $n' \in N$;
- $\lambda' = \lambda\restriction_{Vx'}$, where $\lambda\restriction_{Vx'}$ denotes the restriction of λ to Vx'.

In addition, we define the operator $Update(CT, n, \langle CT_1, \ldots, CT_k\rangle))$ inductively as follows:

- $Update(CT, n, \epsilon)) = CT$;
- $Update(CT, n, \langle CT_1, \ldots, CT_k\rangle)) = Update((CT \bigcirc_{n \cdot i} CT_1), n, \langle CT_2, \ldots, CT_k\rangle))$, where i is the smallest number such that $n \cdot i \notin CT$.

The first condition sets the base case: updating CT at node n with the empty sequence ϵ of control trees leaves CT unchanged. According to the second condition, CT is updated at node n by first inserting the control tree CT_1 in place of the first empty subtree of n, thus obtaining the tree $(CT \bigcirc_{n \cdot i} CT_1)$, and then recursively updating the resulting tree with the remaining sequence of trees.

Finally, we have to suitably account for the data flow induced be the firing of a (compound) transition. For a transition $t \in T_i$, we introduce the notion of *implicit actions* $\alpha_{impl}(t)$ as the set of actions associated with the possible implicit transitions triggered by the execution of t. In particular, if t performs a call operation, namely the target of t is a box or a pair of the form (b, en), the implicit transitions starting from each default node df_M of the machines associated with b by $Y_i(b)$ or from the implicit node en are all executed immediately after t. Therefore, the action performed by each such implicit transition must be executed immediately after the action of the transition t. Examples of such implicit transitions are the transitions *T01* and *T02* of machine *M_MovAuth*, triggered by the firing of transitions *T05* and *T06*, respectively of machine *M_ManageTrain* (see Figs. 3 and 4).

Observe that, when the call box contains more than one machine, several implicit transitions may be triggered by the execution of t. For this reason, the implicit action of t is defined as a set of actions. More formally:

$$\alpha_{impl}(t) = \begin{cases} \left\{ \alpha(t') \;\middle|\; \begin{array}{l} Src_j(t') = df_{M_j} \text{ and} \\ j \text{ occurs in } Y_i(b) \end{array} \right\} & \text{if } Trg_i(t) = b \in Bx_i; \\ \{\alpha(t')\} & \text{if } Trg_i(t) = (b, en) \\ & \quad \text{and } Src_j(t') = en; \\ \emptyset & \text{otherwise.} \end{cases}$$

Notice that, for $t \in T_i$, the set $\alpha_{impl}(t)$ contains actions associated with transitions of another machine, therefore it may still contain non-ground actions, even when M_i is a ground machine.

We can now provide the semantics of compound transitions $ct = \langle\langle t_1, \ldots, t_k\rangle, \langle t'_1, \ldots, t'_d\rangle\rangle$ w.r.t. a state $s = \langle CT, Fr, \theta\rangle$. Assume ct is enabled at s, i.e., there is a set of nodes of the frontier Fr that are all children of the same node $n \in Fr$, and such that $s \models Enabled(ct, N)$. The nodes in N are labelled with the sources of the incoming transitions $\langle t_1, \ldots, t_k\rangle$ of ct. A successor state s' of s is obtained by first removing from CT and from Fr (using the operator *Remove*) all the subtrees of n rooted in the source nodes in N, and by adding (using the operator *Update*) the control trees in $Trees(\langle t'_1, \ldots, t'_d\rangle)$ as additional children of n. These operations take care of modification of the control tree at state s.

To account for the modification data flow at s, we have to consider the effects on the evaluation context θ of the actions of the compound transition ct, which results in the evaluation context θ' of the target state s'. As explained above, the action of a compound transition ct consists of a pair of sets $\langle \alpha^{pre}(ct), \alpha^{post}(ct)\rangle$ of actions associated with the DSTM transitions composing ct, where $\alpha^{pre}(ct)$ is the set of actions performed by the incoming transitions of ct and $\alpha^{post}(ct)$ the set containing those performed by the outgoing DSTM transitions of ct. Clearly, the execution of a compound transition may involve the parallel execution of multiple incoming transitions, possibly followed by the parallel execution multiple outgoing transitions. Hence, we need to take into consideration the different possible orderings of execution of the actions of those transitions. In other words, any permutation of the actions of the incoming transitions followed by a permutation of the actions of the outgoing transitions is a possible outcome for the execution of the action of the compound transition.

Let us consider first a fork compound transition of the form $\langle\langle t\rangle, \langle t'_1, \ldots, t'_k\rangle\rangle$. Then, $\alpha^{pre}(ct) = \emptyset$ (no action is allowed for the incoming transition t) and $\alpha^{post}(ct)$ collects the actions of the outgoing DSTM transitions $\langle t'_1, \ldots, t'_k\rangle$. All those transitions, with the possible exception of the asynchronous fork transition, must be call transitions and, by the constrains of a fork transitions, all the called box must contain a single machine. Let b_j be the box called by transition t'_1 and M_{b_j} the (parametric) machine associated with b_j. On entering that box, either the implicit transition from the default pseudo-node (call by default) or from one of its entering nodes (call by entering) is taken and let t_j be such a transition. The firing of t'_j must perform the action α'_j of transition t'_j immediately followed by the single implicit action contained in $\alpha_{impl}(t'_j) = \{\alpha_j\}$ due to the implicit transition t_j. In other words, the execution of t'_j triggers the execution of the sequence of actions $\alpha'_j; \alpha_j$. Since the transitions $\langle t'_1, \ldots, t'_k\rangle$ are executed concurrently, the sequences of actions executed for $j \in [1, \ldots, k]$ can be executed in any order.

Consider the case where ct is a join compound transition $\langle\langle t'_1, \ldots, t'_k\rangle, \langle t\rangle\rangle$ and t is a call transition. In this case, since no action is associated with the incoming transitions $\langle t'_1, \ldots, t'_k\rangle$, the only possible actions are due to the outgoing call transition t and to the possible implicit transitions, one for each machine associated with the box called by t, if t is indeed a call transition. These implicit transitions are executed in parallel and any permutation of them can occur. As a consequence, the actions executed in this case consist of the action α of transition t followed by any permutation of the implicit actions in $\alpha_{impl}(t)$, associated with those implicit transitions (if t is not a call transition then $\alpha_{impl}(t) = \emptyset$). Similar considerations also apply when ct is a simple compound transition $\langle\langle t\rangle, \langle t\rangle\rangle$, where the DSTM transition t is a call transition.

In all the cases considered above, the implicit actions may still contain parameters, since they are associated with implicit transitions belonging to machines different from the one where the compound transition ct is taken. Therefore, they all must be instantiated with the appropriate parameter instantiation function associated with the call transitions involved.

The intuitions above are formalized by the following definition of transition relation. Given a state $s = \langle CT, Fr, \theta\rangle$ and a compound transition $ct = \langle\langle t_1, \ldots, t_k\rangle, \langle t'_1, \ldots, t'_d\rangle\rangle$ of machine M_i, we have that

$$\langle CT, Fr, \theta\rangle \xrightarrow{ct} \langle CT', Fr', \theta'\rangle \quad \text{iff}$$

there exist a vertex n and a set $N \subseteq Fr$ of children of n in the frontier of s such that:

1. $s \models Enabled(ct, N)$;
2. $CT' = \langle Vx', \lambda'\rangle = Update(CT'', n, Trees(\langle t'_1, \ldots, t'_d\rangle))$ and $CT'' = \langle Vx'', \lambda''\rangle = Remove(CT, N)$;
3. $Fr' = Remove(Fr, N)$;
4. $\theta \xrightarrow{\alpha_1;\alpha_2} \theta'$, where $\alpha_1 = \alpha^{pre}(ct)$ and
   (a) if ct is a fork compound transition of the form $\langle\langle t\rangle, \langle t'_1, \ldots, t'_k\rangle\rangle$, then α_2 is a permutation of the set of actions

$$\left\{ \alpha(t'_j); \alpha[\ell_j] \mid j \in [1, \ldots, k], \alpha_{impl}(t'_j) = \{\alpha\} \text{ and } \ell_j = Inst(t'_j) \right\};$$

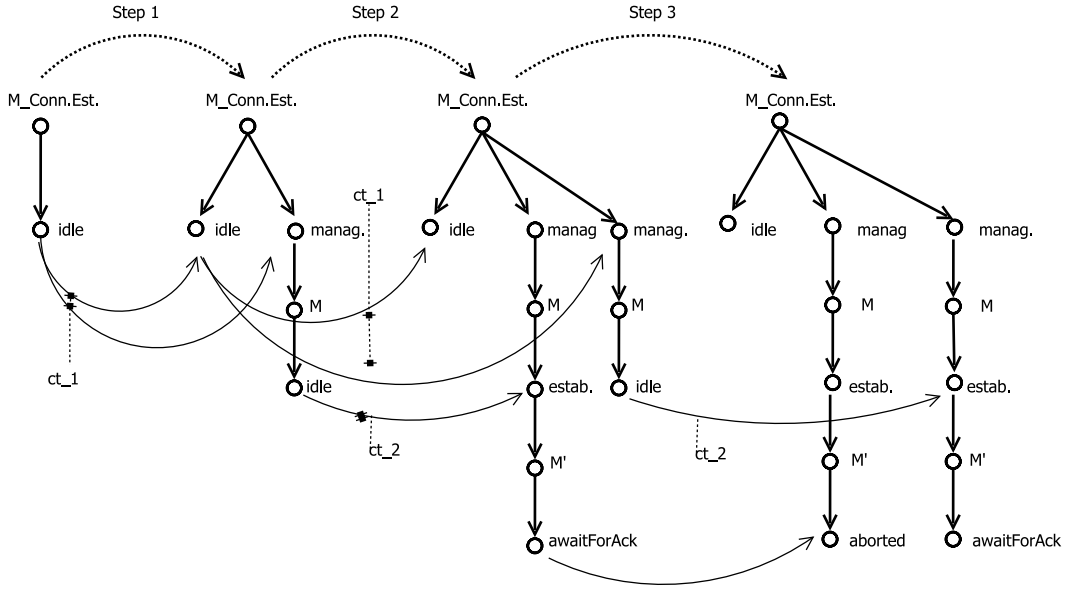

Fig. 9. Simulation of three steps. (M is for *M_ManageTrain* and M' for *M_SessionEstablishment*.)

(b) if ct is a join compound transition of the form $\langle\langle t_1, \ldots, t_k\rangle, \langle t\rangle\rangle$ or a simple transition of the form $\langle\langle t\rangle, \langle t\rangle\rangle$, then $\alpha_2 = \alpha^{post}(ct); \alpha$, where, if t is a call transition to box b, then α is a permutation of the set

$$\left\{ \alpha(\bar{t}_j)[\ell_j] \;\middle|\; \begin{array}{l} \alpha(\bar{t}_j) \in \alpha_{impl}(t),\ j \in [1, \ldots, |Y(b)|],\ Src_z(\bar{t}_j) = df_{M_z}, \\ z = (Y(b))_j \text{ and } \ell_j = (Inst(t))_j \end{array} \right\}$$

and $\alpha = \epsilon$, otherwise.

When no more transitions of the current step can be performed on CT (depending on the values of variables, channels contents, control tree and frontier), the LTS has a transition corresponding to the completion of the current step and the initialization of the next step:

$$\langle CT, Fr, \theta\rangle \xrightarrow{next} \langle CT, Fr', \theta'\rangle$$

where $Fr' = T$, $CT = \langle T, \lambda\rangle$, $\theta = \langle \rho, \chi, \eta\rangle$ and $\theta' = \langle \rho, \chi', \emptyset\rangle$ such that:

- for all $c \in C_I$, $\chi'(c) = \chi(c)$, and
- for all $c \in C_E$, $\chi'(c) \in \{\epsilon\} \cup \mathcal{D}(c)$ and if $\eta(c) \neq \perp$ then $\chi'(c) = \langle \eta(c)\rangle$.

Running example *(part 8).* Fig. 9 exemplifies three distinct steps of the running example. In the initial state machine *M_CommunicationEstablishment* is located in state *idle*. We assume that in that state the compound transition $ct_1 = \langle\langle T03\rangle, \langle T04, T05\rangle\rangle$ is enabled at the set of nodes $\{1\}$, with $\lambda(1) = idle$. The firing of that compound transition generates the two control trees in Fig. 8(a). According to condition 2. above, we first remove the subtree rooted in the vertex labelled *idle* from the control tree, and then append as children of its parent the root ϵ and the two trees in Fig. 8(a). This results in the second control tree from the left in Fig. 9. This completes the step, since both the new added leaves (both labelled by *idle*) do not belong to the frontier. Therefore, a new step is initialized.

In the second step, two compound transitions, namely $ct_2 = \langle\langle T02\rangle, \langle T02\rangle\rangle$ and again ct_1, are enabled. The firing of ct_1, similarly to the previous case, first removes the subtree rooted in the vertex labelled *idle*, and then adds the two control trees in Fig. 8(a) as children of the root. Since the subtree rooted in vertex 2 is in the frontier, transition ct_2 can fire in the same step after the firing of ct_1. The firing of transition ct_2 removes the leaf of the subtree rooted in 2 labelled by *idle*, and appends the control tree in Fig. 8(b) to vertex $2 \cdot 1 \cdot 1$. The resulting control tree is the third control tree in Fig. 9. Notice that the two transitions could be executed in any order. At this point, the second step is maximal, as none of the leaves in the resulting control tree belongs to the frontier and the third step can be initialized.

In the third step we assume that transition ct_1 is not enabled (for instance because its trigger does not hold), whereas transition ct_2 is enabled in the leaf of the path rooted in vertex 3. Moreover, we assume that the compound transition $ct_3 = \langle\langle T02\rangle, \langle T02\rangle\rangle$, with *T02* the transition of machine *M_SessionEstablishment*, is enabled in the leaf labelled by *waitForAck*. As in the previous case, those two transitions can fire in any order, leading to the last control tree in Fig. 9. Now the only leaf of the control tree still in the frontier is the vertex labelled by *idle*. Since we have assumed that the trigger *C_request*?

does not hold, transition *T02* of machine *M_CommunicationEstablishment* cannot fire. Moreover, the compound transition $ct_4 = \langle\langle T06, T07\rangle, \langle T08\rangle\rangle$ is not enabled to fire either, since *T07* is not enabled in the considered state, hence this step is maximal and the next step starts. ■

5. From DSTM to Promela

The DSTM formalism has a great expressive power and can be used in different stages of the system life-cycle, spanning from specification to verification and validation. It can also be used as a source formalism to derive formal models expressed with different lower-level formats, so as to enable property verification and/or test case generation. In particular, we developed an automated process to derive models expressed in Promela, the specification language supported by the Spin model checker [22]. Even though our approach is not tied to any specific technology, we had to take a specific direction within the CRYSTAL project, where our ultimate goal is to develop a process for automated test case generation, exploiting the model checkers' ability to generate counterexamples for violated properties. Among the available model-checking back ends, we chose Spin, as it supports many of the control flow and data flow constructs contained in the DSTM formalism, such as buffered message passing, communication via shared memory and dynamic instantiation of processes. These features make Spin a natural choice to tackle the problem at hand. In addition, Spin is a well known on-the-fly model checker, able to handle efficiently problems of large size.

The objective of this section is to present the main issues underlying the mapping from DSTM to Promela, and explain how the step semantics is realized by means of a dedicated Promela process, called *Engine*, which is also responsible for modelling the non-deterministic external environment of the system under consideration. This section focuses on the translation of the DSTM control-flow, with particular attention to the dynamic instantiation of machines, concurrency issues and machine preemptive termination. The crucial issues concerning the binding of the data-flow constructs with Promela is also sketched. A detailed discussion about this binding is, however, out of the scope of this paper.

We show that the mapping from DSTM machines to Promela is fully automatable. To this end, we propose a chain of model transformations, in part implemented in the ATLAS language [23] and in part in Java. At the time of writing, a first version of the required transformations is available.

The transformation into Promela is performed in two steps. The first step converts a DSTM model into classic (flat) state machines, by getting rid of the hierarchical structure (the vertical modularity) and suitably rewriting machine instantiation, termination and the horizontal modularity (i.e., parallelism). This first step is essential, since Promela does not support hierarchical specifications. The second step, instead, transforms the resulting state machines into an actual Promela model. The two steps are described below.

5.1. Step 1: from DSTM to ordinary state machine

In this step all fork and join pseudonodes and the boxes, which envelop inner machines inside the DSTM model, are removed. The removed elements are substituted by nodes and transitions, which are also used to model activation and termination of machines. Nodes, implicit transitions and internal transitions in the source DSTM are left unchanged, as they correspond to elements that can easily be encoded in Promela.

As we will describe later with more details, each machine is translated into a Promela *process* type, and an instance of a machine corresponds to a Promela process. A unique identifier is associated to each process and recorded at creation time in a variable named *pid*. The identifiers of the DSTM nodes in which the active instances of the same machine *M* currently reside, are recorded in a vector *state_M*, indexed by the process identifiers. The size of such vector is, therefore, equal to the maximum number of processes that can run concurrently in the system.

When removing a box, three different situations may occur, depending upon the source and target of the transitions entering and exiting from the box (see Table 2 in Section 3 for possible cases). Hence, we distinguish three different mapping schemata for boxes:

1. *simple box*: the source of the transition entering the box is either another box or a node;
2. *asynchronous fork*: the source of the transition entering the box is a fork pseudonode and an asynchronous transition exists from the fork pseudonode to a node;
3. *synchronous fork*: the source of the transition entering the box is a fork pseudonode and no asynchronous transition exits the fork pseudonode.

Simple box In this case all the boxes are removed and substituted by nodes having the same name. The transitions entering and exiting from the boxes are substituted with transitions entering and exiting from the corresponding nodes. Such transitions inherit all the component of the decorations of the original transitions. However, in order to model the call and termination operations of the machines associated with the original boxes, such decorations need to be augmented with additional actions and guards.

Let us consider entering transitions first. The guard is preserved as in the original transition. However, two actions need to be added. A first action performs the activation of the machine and involves a special construct *run*, which corresponds to the Promela instruction with the same name that performs process activations. This action has the

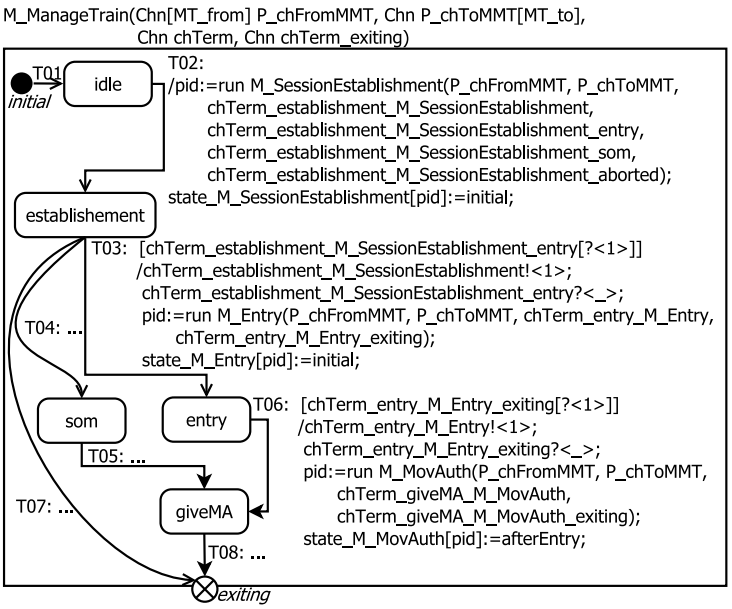

Fig. 10. Flattening of machine *M_ManageTrain*.

form *run Machine_Name*(*parameter list*), where *Machine_Name* is the name of the machine associated with the replaced box. The *run* construct returns a process identifier which is stored in the variable *pid*. The second action is responsible for initializing the initial state of the activated machine (either the default or an entering node) and has the form $state\_MachineName[pid] = init - state$. If more than one machine is associated with a box, a sequence of such actions is used.

This is illustrated in Fig. 10, where the result of the flattening of machine *M_ManageTrain* of Fig. 4, is reported, where the four boxes of that machine are replaced by four nodes.

In Fig. 4 transitions *T02*, *T03*, *T04* and *T05* are *call by default* transitions, whereas *T06* is a *call by entering* transition which specifies the entering node *afterEntry* of machine *M_MovAuth*.

In Fig. 10 only transitions *T02*, *T03* and *T06* are fully detailed, for sake of space.

T02 activates the machine *M_SessionEstablishment*, hence its decoration is modified by adding the action:

$$pid := run\ M\_SessionEstablishment(parameter\ list)$$

In addition, the position *pid* of the vector *state_M_SessionEstablishment* is initialized to the default entering node *initial*. Notice that, as *T06* is a *call by entering* transition, the initial state of the machine *M_MovAuth* is assigned to the entering node *afterEntry*.

The parameter list within a *run* construct contains all the parameters of the original box (as specified by the domain of the instantiation function *Inst* of the box). Additional parameters are, however, needed to correctly model the termination of the called machines. In particular, for each machine with n exit nodes, $n+1$ *internal* channels are introduced. With each exit node of *MachineName* an *exit channel chTerm_BoxName_MachineName_ExitNodeName* is associated. Each of these channels is used by the called process to signal its caller the reaching of the corresponding exit node. Moreover, the additional *termination channel chTerm_BoxName_MachineName* is used by the caller to issue a termination signal to the called *MachineName*, after it has reached an exit node.

Fig. 10 exemplifies the termination of the machine instance *M_SessionEstablishment*, by detailing the *return by exiting* transitions *T03* and *return by default* transition *T06* of Fig. 4. For example, the parameter list of the *run* construct for transition *T02*, instantiating machine *M_SessionEstablishment*, contains four channels: one for each exit node of the machine, plus the channel used by the caller *M_ManageTrain* to send the termination message.

Let us now consider the transitions exiting from a box. Each of them is replaced by a suitable transition exiting from the node replacing the box. If the original transition is a *return by exiting*, then the transition must be taken only if the called machine has terminated. Therefore, the decoration of the replacing transition must also check that an exit message has been sent over the channel associated with that exit node *ExitNodeName*. Hence, the guard *chTerm_BoxName_MachineName_ExitNodeName*? $< 1 >$ is added in conjunction to the guard of the original replaced transition. If, on the other hand, the transition is a *return by default*, then it can be taken as soon as the called process has reached any exit node. In this case, the guard of the transition is augmented with the disjunction of all the guards of the form *chTerm_BoxName_MachineName_ExitNodeName*? $< 1 >$, one for each exit node. In either case, when the transition is taken, the called must terminate. This is achieved by executing an action where the caller sends a termination message over channel *chTerm_BoxName_MachineName* to the called process.

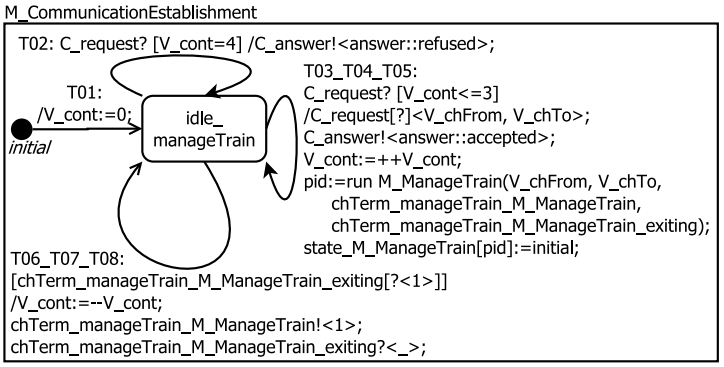

Fig. 11. Flattening of machine *M_CommunicationEstablishment*.

In Fig. 10 a condition is added on transition *T03* that checks for the reception of the notification message from *M_CommunicationEstablishment* over the channel associated to the exit node *entry*. Two actions are added on *T03*: the first models the transmission of the termination message from *M_ManageTrain* to *M_SessionEstablishment*, the latter is needed to consume the message on the same channel (as it is an internal channel). The same holds for transition *T06*.

Notice that, as *T06* is a *return by default* transition, all the exit nodes of the terminating machine must be considered. In this specific case machine *M_Entry* has just one exit node and only the condition *chTerm_entry_M_Entry*$[? < 1 >]$ is added to the guard.

Asynchronous fork In case of an asynchronous fork, the calling process continues to run concurrently with the invoked machines. Since, by well-formedness (see Section 3), every join transition is always paired with a previous fork transition, we can consider each pair of fork and join transitions as a single block. In addition, the control-flow between the two transitions is entirely contained within the boxes entered by the fork, i.e., the fork enters a set boxes and the return transitions from those boxes lead directly to a corresponding join. Remember that a transition exiting from a fork pseudonode can specify neither trigger nor condition, it may, however, perform an action. According to the DSTM semantics (see Section 4), the actions associated with the transitions exiting from a fork pseudonode can be executed in any order.

Therefore, the entire block, containing the fork, a corresponding join and the called boxes, is removed and substituted by suitable *internal* transitions. Some of these *internal* transitions model the fork operation and lead from the source node of the fork to the target of the asynchronous fork transition. These transitions take care of modelling the possible permutations of the actions associated with the transitions exiting from the fork pseudonode (see case 4.a in the semantic of compound transitions in Section 4). Similarly, to model the join operation we need to add one or more internal transitions, depending on whether the considered join is non-preemptive or preemptive. Each such transition leads from the source node of the fork to the target of the join.

Let us consider the transitions modelling the fork first. The decoration of each such new transition is obtained, similarly to the case of a *simple box* described above, by keeping the same guard and trigger of the entering fork transition, and adding to the actions the corresponding *run* constructs that activate the called machines.

This situation is exemplified in Fig. 11 for machine *M_CommunicationEstablishment*, described in detail in Section 3.1 and depicted in Fig. 2. In this case, box *M_ManageTrain* is removed and transitions *T03*, *T04* and *T05* are replaced by the single transition *T03_T04_T05* from the source node of the *internal entering fork* transition *T03* to the target node of the *internal asynchronous fork* transition *T04* exiting from the fork. As node *idle* is both the source and the target node of this transition, the new transition forms a self-loop on that node. The trigger and the condition of *T03* are inherited by transition *T03_T04_T05*. The action on the new transition includes all the actions on the original transitions and, in addition, the actions needed for the activation of the machine inside the box. Since only transition *T03* has an associated action, a single new transition is required in this case. Notice that node *idle* in Fig. 2 has been renamed *idle_manageTrain* in order to keep track of the removed box.

Remember that an *exiting join* transition can specify neither a trigger nor a condition, but it may have an associated action. On the contrary, no transition entering the join can have actions.

If the join is non-preemptive, it is replaced by a single *internal* transition with no trigger and a guard checking for the termination of all the machines associated with the joined boxes. The action of the transition must include the action of *exiting join* transition as well as the actions necessary to deal with the termination of the joined boxes, similarly to the *simple box* case described above.

In case of preemptive join, namely if at least one preemptive *entering join* is present, each *entering join* can trigger the termination of all the other parallel machines, regardless of their current state. In this case, we need a distinct *internal* transition for each preemptive *entering join*. The *internal* transition inherits the same trigger as the original *entering join*. Guards and actions are treated as in the non-preemptive case.

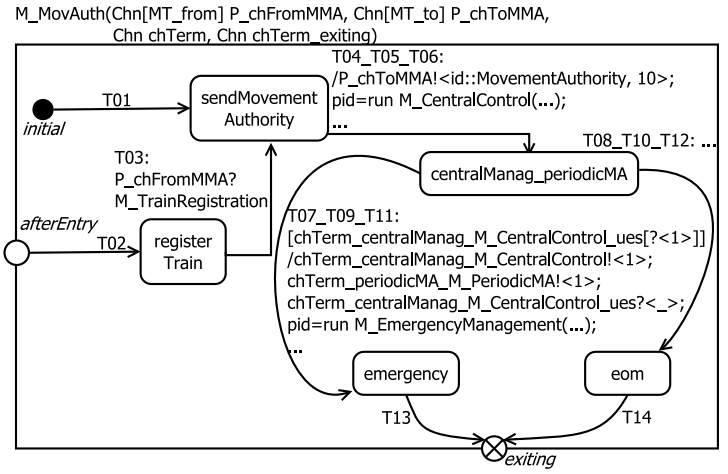

Fig. 12. Flattening of machine *M_MovAuth*.

In Fig. 11, transition *T06_T07_T08* replaces the non-preemptive join transitions *T06*, *T07* and *T08*. The actions on this new transition include those specified on the *exiting join* transition *T08*. The guard is *chTerm_manageTrain_M_ManageTrain_exit*[? < 1 >], checking for the reception of the exit message from machine *M_ManageTrain*. The action associated with transition *T06_T07_T08* is the action of transition *T08* (*V_Cont* := − − *V_Cont*) followed by the consumption of the exit message from *M_ManageTrain* and the sending of the termination message to*M_ManageTrain* along channel *chTerm_manageTrain_M_ManageTrain*.

Synchronous fork In case of a synchronous fork the currently executing process is suspended to wait for the termination of the called machines. In this case, the entire block, containing the fork, the join and the boxes, is substituted with a new node (called *wait* node), which models the state of the suspended calling process. In addition, the transitions in the block are replaced by a set of *internal* transitions to and from the *wait* node.

The *internal* transition, modelling the fork operation from the source of the fork to the *wait* node, is defined as in the case of the *asynchronous fork* illustrated above. The ones from the wait node to the target of the join, modelling the join operation, are defined precisely in the previous case.

This construction is exemplified in Fig. 12 on machine *M_MovAuth*, depicted in Fig. 3 and described in detail in Section 3.1. The state machine in the figure is obtained by removing the two boxes that activate machines *M_CentralControl* and *M_PeriodicMA*. Those boxes are replaced by the *wait* node named *centralManag_periodicMA*.

Transitions *T04*, *T05* and *T06* are replaced by transition *T04_T05_T06*, whose decoration is not completely reported in the figure. Its action performs the instantiation of the two machines *M_CentralControl* and *M_PeriodicMA* by means of suitable *run* constructs, according to the mapping rules explained above.

Transitions *T07*, *T09* and *T11* are replaced by the *internal* transition *T07_T09_T11*, while transitions *T08*, *T10* and *T12* are replaced by transition *T08_T10_T12*.

The example allows us to illustrate an instance of a preemptive join. In fact, the transition *T07* (resp., *T10*) in Fig. 3 is preemptive. The guard of transition *T07_T09_T11* (resp., *T08_T10_T12*) holds when the exit message is received from the machine *M_CentralControl* (resp., *M_PeriodicMA*) on channel *chTerm_centralManag_M_CentralControl_ues*. The suspended caller wakes up and sends the termination message to the machines *M_CentralControl* (on *chTerm_centralManag_M_CentralControl*) and *M_PeriodicMA* (on *chTerm_periodicMA_M_PeriodicMA*).

5.2. Step 2: Promela encoding

Below we describe how the data flow and the control flow of the flat model, obtained in the previous section, is translated into Promela.

Translating datatypes Table 4 reports the mapping between DSTM types, variables and channels into their Promela equivalents. The mapping of most of the DSTM types and variables is quite straightforward, the only exception is the translation of DSTM channels which are mapped into global Promela channels.

As for internal channels, the size of the buffer is equal to the *bound* of the corresponding DSTM channels. The type of the messages conveyed on internal channels, with the exception of multi-type channels, is mapped onto the corresponding Promela type. Each multi-type internal channel, instead, is modelled by a set of Promela channels, one for each simple type contained in the multi-type. The set of Promela channels associated with a multi-type channel is managed so as to ensure that at each instant at most one of them contains a message.

Table 4
Mapping of types, variables and channels.

DSTM	Promela equivalent
Data types	
tInteger	*int*
tEnum	$mtype = \{literals_0, literals_1, ..., literals_n\}$;
Variables	
tInteger variable	*int v*;
tEnum variable	*mtype v*;
tChannel variable	*chan v*;
Channels	
external channel of tInteger	*chan c* = [2] *of* {*bit*, *int*};
internal channel of tInteger	*chan c* = [*bound*] *of* {*int*};
external channel of tEnum	*chan c* = [2] *of* {*bit*, *mtype*};
internal channel of tEnum	*chan c* = [*bound*] *of* {*mtype*};
external channel of tCompound	*chan c* = [2] *of* {*bit*, ...};
internal channel of tCompound	*chan c* = [*bound*] *of* {...};
external channel of tMultiType	*chan c_ST1* = [2] *of* {*bit*, ...}; *chan c_ST2* = [2] *of* {*bit*, ...}; ...
internal channel of tMultiType	*chan c_ST1* = [*bound*] *of* {...}; *chan c_ST2* = [*bound*] *of* {...}; ...

As for external channels, according to the DSTM semantics they shall maintain both the message valid in the current step execution and the message to be delivered in the next step (if produced during the execution of the current step). Hence, external channels are translated into Promela channels whose buffer size is 2: the first position is used to store the current message, the second position is used to store the message to be delivered in the next step. In order to avoid that a message produced by the system during the current step could be read in the same step by other machines, the first position of each external channel must be always filled. At this aim a fake message (0) is used if the first position does not contain a valid message. By this mechanism, possible messages produced by a machine during a step are surely stored in the *second* location of external channels through the *send* statement of Promela and they are not sensed during the execution of the current step.

The engine process and the step semantics In order to obtain a suitable Promela specification, we have to correctly implement the step semantics and the firing of transitions according to the *enabledness* condition defined in Section 4 for compound transitions. This condition requires that for a transition of an active machine to fire, none of its ancestors in the control tree must be currently enabled to interrupt the execution of the machine (either by an interrupt transition or by a preemptive join). This condition will be guaranteed by imposing an execution priority among the machines, which is compliant with the hierarchical activation of machines. We also recall that the step semantics prevents sequential firing of transitions within the same execution step. Since in the flat model compound transitions have been removed, we only need to guarantee that at most one enabled transition can fire for each active process.

Each Promela process, modelling a DSTM machine, owns a token in order to fire an enabled transition. When a process owning a token is scheduled, it consumes its token and:

- if none of its transitions is enabled, the process generates and sends a token to each of its children;
- otherwise, an enabled transition is selected and executed.

At the beginning of each step only the process corresponding to the initial machine owns the token. The token propagation guarantees both that the *enabledness* condition mentioned above is correctly simulated and that the sequential firing of transitions within the same step, ensured by the notion of frontier in the semantics, is prevented by the encoding. The non-deterministic nature of the Spin scheduler ensures that all the feasible steps allowed by the DSTM semantics are preserved.

A special Promela process, called *Engine*, is in charge of (i) simulating the test environment by conveying over the external channels messages generated by the environment or by the processes during the previous step, and (ii) starting the token propagation mechanism for each step.

The execution of a single step is then performed by iterating the following two phases:

1. **Step execution.** Each process is scheduled and if it owns the token, then it is allowed to execute at most one of its enabled transitions. If more transitions are enabled, one of them is non-deterministically chosen. This phase ends when all tokens have been consumed: according to the *run-to-completion* semantics the model of the system reaches a stable state in which no transition can be executed without an input change.

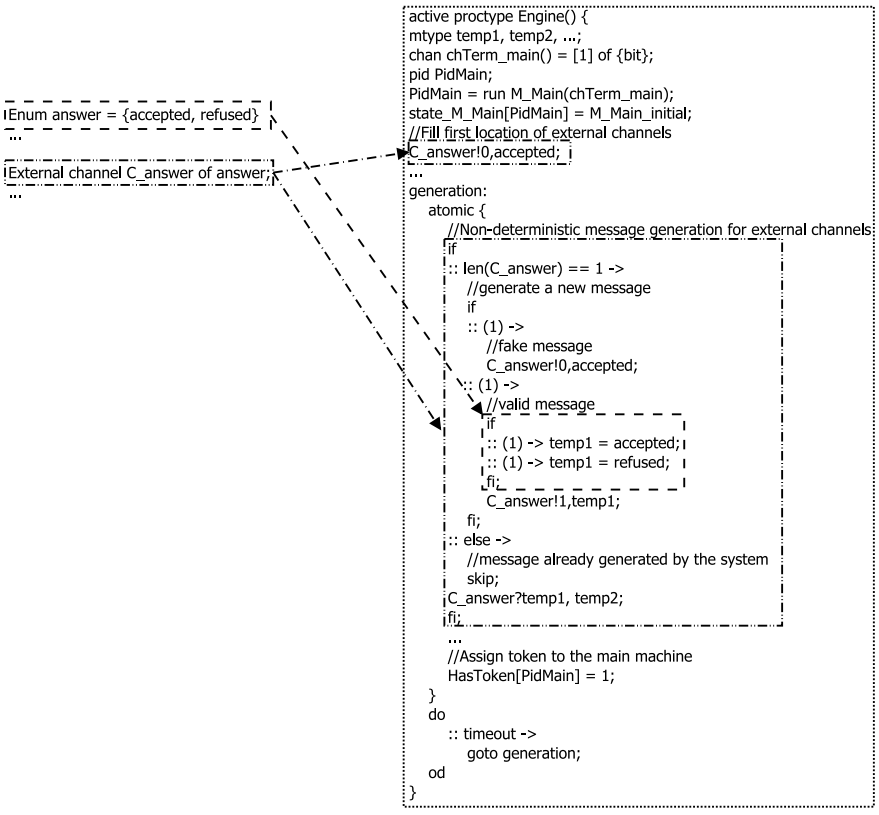

Fig. 13. Promela template of the *Engine* process.

2. **Next step initialization.** The *Engine* process removes all the messages contained in the external channels. Then it inserts in the external channels the messages sent over those channels by the processes in the currently terminating step. It may also generate new messages (whose values are non-deterministically chosen) to insert in external channels still empty, so as to simulate the behaviour of the external environment. Finally, it generates a new token and sends it to the initial machine, thus enabling the execution of the next step.

Engine is the first process of the Promela model to be activated and it is the only process required to be running in the initial model state. As it always performs the two phases just described, it can be automatically generated by instantiating the Promela template shown in Fig. 13 with the channels and the data definition provided by the system model.

First, the *Engine* process runs the main machine and stores its *pid* in the local variable *PidMain* ($PidMain = run\ M\_Main(chTerm\_main);$) and initializes the external channels with dummy values (this operation is exemplified by the channel *C_answer* in Fig. 13). Then, the *Engine* process starts an atomic block (after the label *generation*), in which it non-deterministically generates the messages to be sent over the external channels. As explained above, the evolution of the processes is driven by a suitable message-handling mechanism, which guarantees that during the execution of each step a suitable message is always stored in the first position of the external channels. This is exemplified for the channel *C_answer* by the nested if-constructs in Fig. 13. In detail, process *Engine* checks if *C_answer* only stores one message ($len(C\_answer) == 1$). In this case, the second position is empty and a new message is non-deterministically generated by the environment (which can choose between an empty and the valid message). Otherwise, the second position already contains a message, produced by the system, to be delivered in the next step ($else -> skip;$).

At this point, process *Engine* consumes the message stored in the first position ($C\_answer?temp1,temp2;$). This moves the message available for the next step to the first position of the channel. The generation block ends by assigning the token to the main process ($HasToken[PidMain] = 1;$).

Finally, the *Engine* process enters the last *do* construct, where it keeps waiting until the Promela global variable *timeout* evaluates to true. This occurs when the system reaches a stable state, in which no transition of the current step can be executed. In this case, the *Engine* jumps to the *generation* label, starting a new execution step (*goto start*).

Encoding a single machine in Promela Fig. 14 shows a fragment of the translation for the machine *M_ManageTrain* in Fig. 4. Each machine is translated into a Promela *proctype*. In this way, each instantiated machine corresponds to a distinct Promela process.

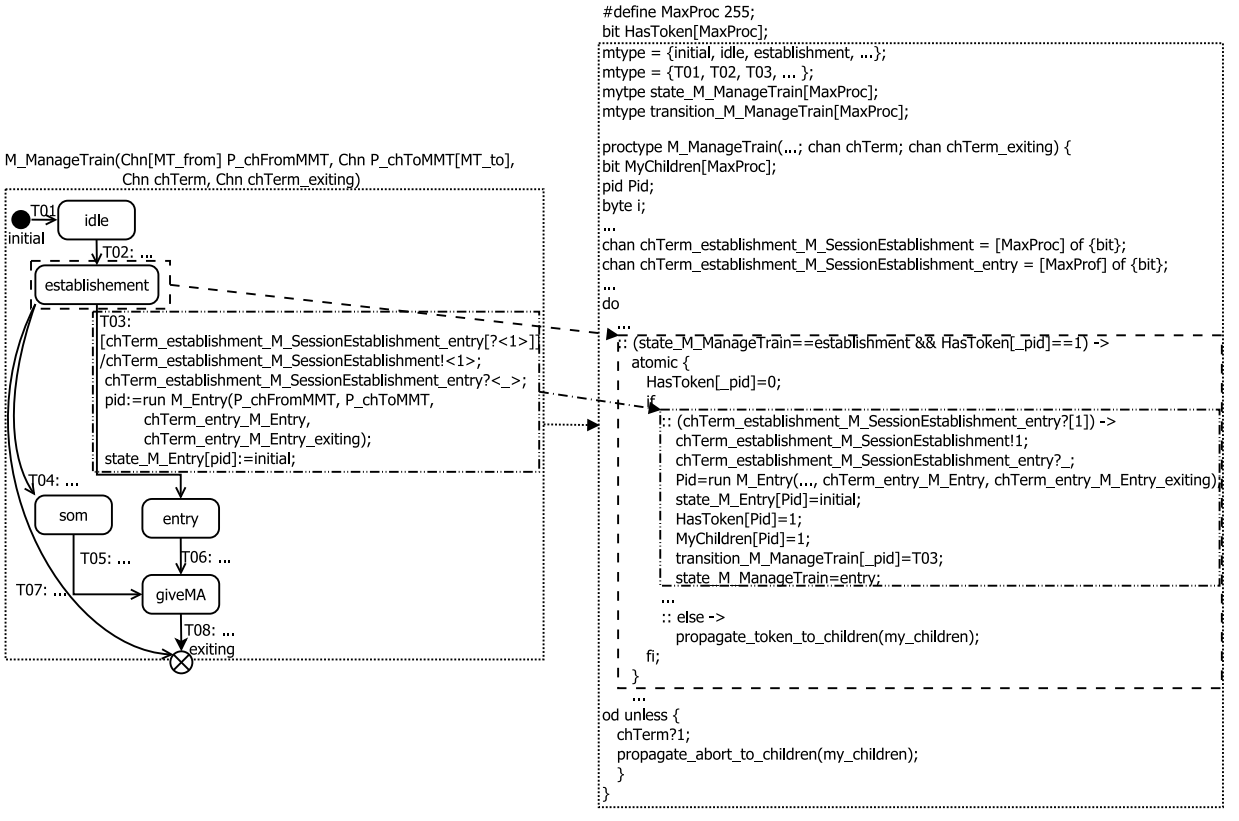

Fig. 14. Control flow mapping.

Two enumeration types (declared as *mtype* in Promela) allow for the introduction of symbolic names for nodes and transitions. Specifically, the first *mtype* declaration in Fig. 14 lists all the names of the nodes of the machine *M_ManageTrain* and the second one lists all the names of the transitions. For each machine, two global variables, *state_M_ManageTrain* (already mentioned in the previous section) and *transition_M_ManageTrain* both typed as *mtype* array, are used to store, in their i-th position, the current state and the last fired transition of the Promela process whose *_pid* is equal to i.

A global variable *HasToken*, typed as a bit array, is used to simulate the step semantics as described above. Specifically, the i-th position of this array is set to 1 if the process, whose *_pid* is equal to i, currently holds the token, meaning that it can evolve. Note that this array is unique for the entire Promela model.

The body of the process consists in an iteration (a *repetition construct—do*) repeated until the termination message is received from the calling process over the channel *chTerm* (exchanged as parameter). The termination message is then propagated to all the children of the current process, if any. In Fig. 14, this propagation mechanism is abstracted into the operation *propagate_abort_to_children*(*my_children*).

Each node of the machine is translated into a *guarded statement*. The guard is satisfied if the specified node is the current state of the process and the process holds the token. In Fig. 14 the guarded statement corresponding to the state *establishment* is reported as an example; the guard is verified if the current state is *establishment* (*state_M_ManageTrain* == *establishment*) and *HasToken*[*_pid*] == 1. The *atomic statement* associated to the guard contains a sequence of statements executed indivisibly. The first statement in the sequence consumes the token (e.g., *HasToken*[*_pid* = 0 in Fig. 14). Then a conditional statement (a *selection construct—if*) contains one guarded statement for each transition exiting from that node, where the guard corresponds to the enabling condition of the transitions (presence of the trigger and truth of the condition) and the associated statements translate the actions (if any). Each action is translated into basic Promela statements and operators, and executed when the associated guarded statement is selected for the execution. If more than one guarded statement is executable, one of them is non-deterministically selected. The *else* branch in the conditional statement guarantees from one hand that the process is not blocked if no transition can fire, from the other hand it executes a block of statements which propagates the token to the children of the current process.

The guarded statement resulting from the translation of the transition *T03* is shown: the guard translates the condition and the related statement translates the action, which performs the termination of the machine *M_CommunicationEstablishment* and the activation, via a *run* statement, of machine *M_Entry*. Additional statements properly update the global variables *transition_M_ManageTrain* and *state_M_ManageTrain*.

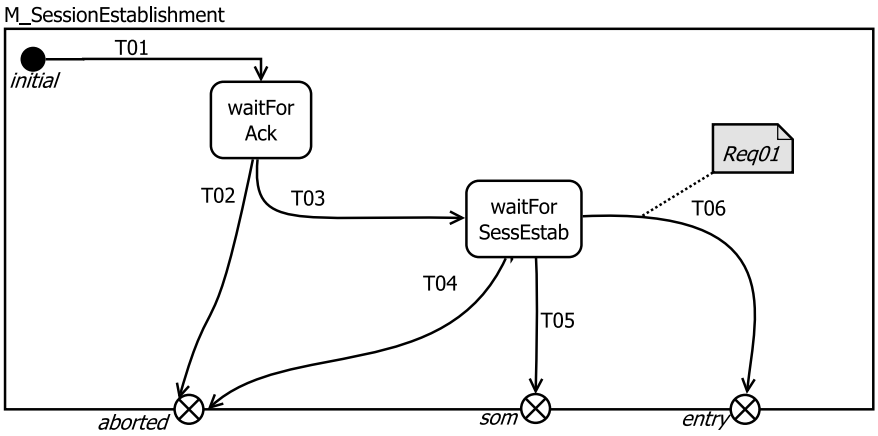

Fig. 15. *M_SessionEstablishment* with the annotation of *Req01*.

6. Application to test case generation

As mentioned in Section 1 and in Section 2, DSTM is a high-level formalism, devised to model system-level specifications of control systems. Starting from a DSTM specification, different transformation chains can be applied with the objective of deriving artefacts of interest by an automated approach. The application described in this section has the objective of automatically generating test cases starting from DSTM models of railway control systems. Specifically, this section describes how functional test cases, which lead the system to cover a specific transition or node of its DSTM specification, can be generated by applying the two steps transformation described in the previous section.

In the present work, we consider the generation of test cases for transition coverage only, as it was the main requirement by ASTS in the CRYSTAL project. Different classes of test cases can actually be tackled by the framework [14]. In order to trace the coverage of transitions back to the specification requirements, transitions and nodes of a DSTM model are annotated with strings representing the requirements' identifiers and, therefore, the generation of a test case covering that element is requested.

A transition *coverage* request is translated into a Promela *never claim*, which models a linear time temporal logic formula. A *never claim* involving some model element (e.g., a transition), requires that the crossing of that model element must never occur. The *never claim* uses two global variables (i.e., *LastTransition* and *LastNode*, typed as *mtype*) that must be specified into the Promela model. These variables record the names of the last fired transition and the last entered node, respectively, in the current execution. The two variables are automatically updated contextually to the execution of a statement encoding the firing of a transition. The *never claim* defines a behaviour in which the variable *LastTransition* (or *LastNode*) never assumes the value of the element to cover. When an execution does not satisfy the *never claim*, a counterexample of the violated property is produced. If the *never claim* is satisfied by all the executions of the model, the test case is infeasible, namely the model element of interest is unreachable from the initial node of the main machine. Without loss of generality, we assume uniqueness of each model element name.

Listing 1 Never claim.

```
never {
step1: if
       :: (LastTransition == M_SessionEstablishment_T06) -> goto endStep
       :: else -> goto step1
       fi;
endStep: skip
}
```

As an example, Listing 1 reports a *never claim*, requiring that variable *LastTransition* never assumes the value *M_SessionEstablishment_T06*. This *never claim* can easily be generated starting from a DSTM model, where the transition *T06* of the machine *M_SessionEstablishment* is annotated with a requirement (i.e., *Req01*), as shown in Fig. 15. The possible counterexample generated by Spin corresponds to a trace that leads to the firing of the specified transition.

We annotated the DSTM model of the *Communication Management* functionality with the requirement identifiers provided by ASTS. Each transition has been annotated with at least one requirement, hence, the final test suite requires the coverage of all the 43 transitions of the case study, numbered from 0 to 42. In the next subsections we report the results obtained on three requirements, as well as the related test cases generated by the Spin model checker. In order to assess the feasibility of the approach, we also report the execution time and the state-space size for each of the 43 test cases.

Table 5
Req01: test case.

Step	*Machine*	*Current values*
0	*M_CommunicationEstablishment*	current node: *initial* firing transition: *T01*
1	*M_CommunicationEstablishment*	current node: *idle* firing transition: *T03_T04_T05* run of *manageTrain_M_ManageTrain*
	M_ManageTrain	current node: *initial* firing transition: *T01*
2	*M_CommunicationEstablishment*	current node: *idle*
	M_ManageTrain	current node: *idle* firing transition: *T02* run of *establishment_M_SessionEstablishment*
	M_SessionEstablishment	current node: *initial* firing transition: *T01*
3	*M_CommunicationEstablishment* *M_ManageTrain*	current node: *idle* current node: *establishment*
	M_SessionEstablishment	current node: *waitForAck* firing transition: *T03*
4	*M_CommunicationEstablishment* *M_ManageTrain*	current node: *idle* current node: *establishment*
	M_SessionEstablishment	current node: *waitForSessEstab* firing transition: *T06*

6.1. Requirements and test cases

For the sake of the presentation, we discuss here only three of the 43 requirements included in the case study. Two of them (namely, *Req01* and *Req02*) result in test cases spanning multiple machines. The last requirement (i.e., *Req03*) results in a test case involving the firing of *asynchronous fork* transitions, which, in turn, requires the dynamic instantiation of machines. The selected requirements are defined as follows:

1. *Req01*: when receiving the *SessionEstablished* message from a train and if the area field of the received message is *L1*, RBC must perform the *Entry* action;
2. *Req02*: when receiving the End of Mission message (EoM) from a train, RBC must initiate the de-registration of the train by performing the *EoM* procedure;
3. *Req03*: when receiving the *connection request* message from a train and if the RBC has already reached the maximum number of connections, it must answer with a *connection refused* message.

Transition *T06* of machine *M_SessionEstablishment* has been annotated with *Req01* (see Fig. 15), Similarly, transition *T10* of the machine *M_MovAuth* has been annotated with *Req02*, and transition *T02* of machine *M_CommunicationEstablishment* with *Req03*. The three test cases, generated according to these requirements, are reported in Table 5 (*Req01*), in Table 6 (*Req02*) and in Table 7 (*Req03*), respectively.

These test cases describe the evolution of the model from the initial node of the initial machine (i.e., the main machine *M_CommunicationEstablishment*) to the firing of the transition annotated with the corresponding requirement. In order to generate the test cases, a complete DSTM specification is required. Nevertheless, the *Communication Management* functionality depends on other procedures, whose formal definition is out of the scope of this work. For this reason, the DSTM model has been completed by substituting these procedures with *stub* machines containing a single node, named *working*, an entering and an exit node that correspond to the interface of the specific machine. Hence, machines *M_Entry*, *M_StartOfMission*, *M_CentralControl*, *M_PeriodicMA*, *M_EmergencyManagement*, *M_EndOfMission* are modelled by stub machines. Notice that, the mapping from DSTM to Promela has been defined to preserve the information needed to trace back the elements of the counterexample generated by Spin to the ones of the original DSTM model.

In fact, *step 1* in Table 5 reports a complete execution involving the firing of transition *T03_T04_T05* in the machine *M_CommunicationEstablishment*, where the asynchronous fork pseudonode is crossed four times. The test case in Table 7 is obtained as an execution path from the initial node to the transition *T02* of the initial machine, which fires only if a connection request is refused (i.e., only when four connections have already been granted). This requires that four instances of machine *M_ManageTrain* have been instantiated and are currently running (*steps 1, 2, 3, 4* in Table 7).

Table 6
Req02: test case.

Step	*Machine*	*Current values*
0	*M_CommunicationEstablishment*	current node: *initial* firing transition: *T01*
1	*M_CommunicationEstablishment*	current node: *idle* firing transition: *T03_T04_T05* run of *manageTrain_M_ManageTrain*
	M_ManageTrain	current node: *initial* firing transition: *T01*
2	*M_CommunicationEstablishment*	current node: *idle*
	M_ManageTrain	current node: *idle* firing transition: *T02* run of *establishment_M_SessionEstablishment*
	M_SessionEstablishment	current node: *initial* firing transition: *T01*
3	*M_CommunicationEstablishment*	current node: *idle*
	M_ManageTrain	current node: *establishment*
	M_SessionEstablishment	current node: *waitForAck* firing transition: *T03*
4	*M_CommunicationEstablishment*	current node: *idle*
	M_ManageTrain	current node: *establishment*
	M_SessionEstablishment	current node: *waitForSessEstab* firing transition: *T05*
5	*M_CommunicationEstablishment*	current node: *idle*
	M_ManageTrain	current node: *establishment* firing transition: *T04* run of *som_M_StartOfMission*
	M_StartOfMission	current node: *initial* firing transition: *T01*
6	*M_CommunicationEstablishment*	current node: *idle*
	M_ManageTrain	current node: *som*
	M_StartOfMission	current node: *working* firing transition: *T02*
7	*M_CommunicationEstablishment*	current node: *idle*
	M_ManageTrain	current node: *som* firing transition: *T05* run of *giveMA_M_MovAuth*
	M_MovAuth	current node: *initial* firing transition: *T01*
8	*M_CommunicationEstablishment*	current node: *idle*
	M_ManageTrain	current node: *giveMA*
	M_MovAuth	current node: *sendMovementAuthority* firing transition: *T04_T05_T06* run of *periodicMA_M_PeriodicMA* run of *centralManag_M_CentralControl*
	M_PeriodicMA	current node: *initial* firing transition: *T01*
	M_CentralControl	current node: *initial* firing transition: *T01*
9	*M_CommunicationEstablishment*	current node: *idle*
	M_ManageTrain	current node: *giveMA*
	M_MovAuth	current node: *centralManag_periodicMA*
	M_PeriodicMA	current node: *working* firing transition: *T03*
	M_CentralControl	current node: *working* firing transition: *T02*
10	*M_CommunicationEstablishment*	current node: *idle*
	M_MovAuth	current node: *centralManag_periodicMA* firing transition: *T08_T10_T12*

Table 7
Req03: test case.

Step	*Machine*	*Current values*
0	*M_CommunicationEstablishment*	current node: *initial* firing transition: *T01*
1	*M_CommunicationEstablishment*	current node: *idle* firing transition: *T03_T04_T05* run of *manageTrain_M_ManageTrain*
	M_ManageTrain	current node: *initial* firing transition: *T01*
2	*M_CommunicationEstablishment*	current node: *idle* firing transition: *T03_T04_T05* run of *manageTrain_M_ManageTrain*
	M_ManageTrain	current node: *idle*
	M_ManageTrain	current node: *initial* firing transition: *T01*
3	*M_CommunicationEstablishment*	current node: *idle* firing transition: *T03_T04_T05* run of *manageTrain_M_ManageTrain*
	M_ManageTrain	current node: *idle*
	M_ManageTrain	current node: *idle*
	M_ManageTrain	current node: *initial* firing transition: *T01*
4	*M_CommunicationEstablishment*	current node: *idle* firing transition: *T03_T04_T05* run of *manageTrain_M_ManageTrain*
	M_ManageTrain	current node: *idle*
	M_ManageTrain	current node: *idle*
	M_ManageTrain	current node: *idle*
	M_ManageTrain	current node: *initial* firing transition: *T01*
5	*M_CommunicationEstablishment*	current node: *idle* firing transition: *T02*
	M_ManageTrain	current node: *idle*
	M_ManageTrain	current node: *idle*
	M_ManageTrain	current node: *idle*
	M_ManageTrain	current node: *idle*

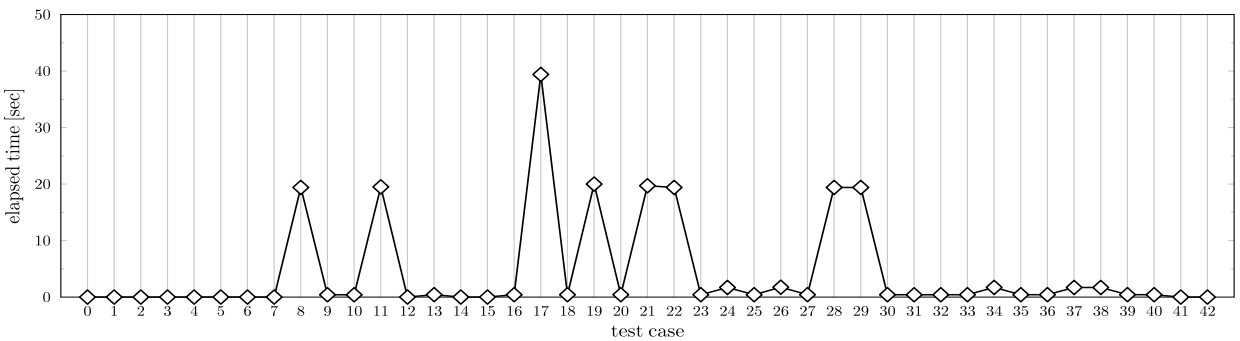

Fig. 16. Execution times.

6.2. Execution times

The analysis of the execution times has been performed by generating 43 test cases covering all transitions of the DSTM model. For each test case, the generation time and the size of the explored state space have been collected.

No advanced Spin optimization techniques have been applied, except for the *safety* compiling directive, that uses less memory in case no cycle detection is needed. The experiments have been executed on a laptop computer, equipped with an Intel Quad-core i7-2670QM CPU 2.20 GHz and 8 GB of RAM.

The execution times (in seconds) are reported in Fig. 16, where the x-axis reports the test case identifiers (from 0 to 42). Most of test cases were generated in about 1 second, while eight of them of the test cases required between 20 to

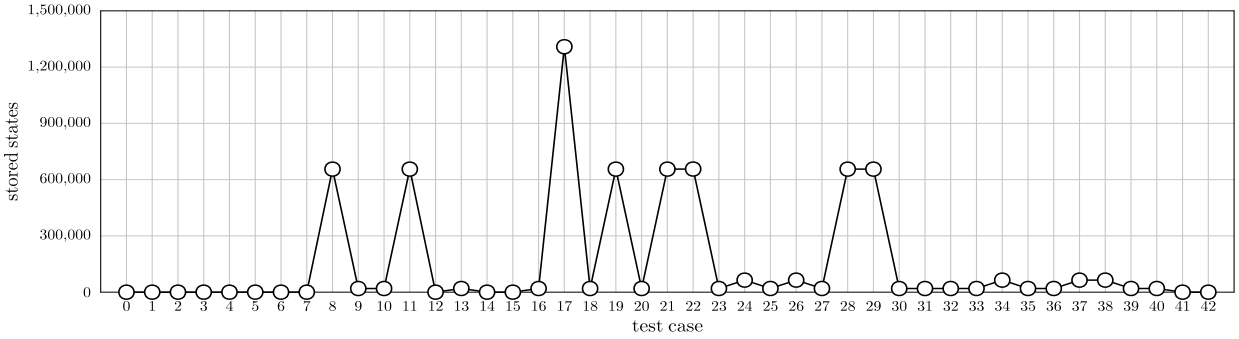

Fig. 17. Number of stored states.

40 seconds. Fig. 17 reports the number of states explored by Spin during the generation of the corresponding test case. The maximum number of stored states (1.3M) has been obtained for the generation of test case 17, and corresponds to the maximum execution time of 40 seconds. The higher generation time and state-space size for some of the test cases seems to be connected to the degree of communication with the external environment (i.e., with the trains) required to reach the firing of the corresponding transition. This, in turn, results in a higher number of non-deterministic choices for the simulation of the messages received from the trains, and, therefore, a higher degree of backtracking performed by the model checking procedure. Nonetheless, the results are very encouraging, since the automatic generation of the entire suite of test cases required about 3 minutes.

7. Conclusions

In this paper we have presented a framework for the specification of the behaviour of critical control systems and the automated generation of test cases for black-box system-level testing using model checking techniques. The critical nature of the systems to model and the high-level of usability required by the application domain suggested: (1) a strong formalization of the language; (2) the synthesis and the extension of some of the features of existing FSM-based languages, and (3) the capability to be integrated into modern model-based processes. The proposed specification formalism is Dynamic STate Machines (DSTMs), a new class of hierarchical state machines to be used in model-based verification contexts. The language has its main strengths in the extended semantics of fork and join which allows for the dynamic instantiation of parametric machines and their preemptive termination. The proposed language has been endowed with a formal definition of syntax and semantics. A prototypical implementation (actually an executable semantics) has been defined in the model checking environment Spin by using the executable specification language Promela. The generation of test cases exploits the well understood technique of counterexample generation for never claims. The modelling approach and the test generation has been applied to a modern railway control system in order to exemplify the natural expressiveness of the specification language and its potentialities for test generation. The prototypical translation of specifications in Promela deserves further steps: (i) optimization techniques have to be introduced to reduce the semantics state space (ii) performance and scalability issues have to be systematically investigated (iii) more structured forms of test case generation will be considered. Moreover we are currently designing a translation of the DSTM models in an alternative model checker environment, NuSMV [13]. Such a translation will allow us to compare the performance obtained by applying different model checking techniques and tools.

Acknowledgements

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

Appendix A. Data flow semantics

We now provide the notion of trigger and guard satisfaction w.r.t. an evaluation context θ and the change of evaluation context due to the execution of an action.

Evaluation of ground transition decorations A satisfiability relation of a ground trigger ξ w.r.t. $\theta = \langle \rho, \chi, \eta \rangle$, written $\theta \models \xi$, is recursively defined as follows:

- $\theta \models \tau$;
- $\theta \models c?$, with $c \in C_E$, if $|\chi(c)| = 1$ (the channel is not empty);

- $\theta \models c?T_i$, with $c \in C_E$, if $|\chi(c_i)| = 1$, $head(\chi(c)) \in \mathcal{D}(T_i)$ and either $type(c) \in \mathbf{MT}$, $T_i \in type(c)$, or $type(c) \in \mathbf{ST}$ and $T_i = type(c)$;
- $\theta \models \xi_1 \wedge \xi_2$ if $\theta \models \xi_1$ and $\theta \models \xi_2$;
- $\theta \models \xi_1 \vee \xi_2$ if $\theta \models \xi_1$ or $\theta \models \xi_2$;
- $\theta \models \neg\xi$ if it is not the case that $\theta \models \xi$.

Both actions and guards contain terms. Therefore, we need to preliminary introduce the evaluation $[\![trm]\!]_\theta$ of a ground term trm w.r.t. an evaluation context $\theta = \langle \rho, \chi, \eta \rangle$, namely a function $[\![\cdot]\!]_\cdot : trm_P \times \Theta \to \mathcal{D}(\mathbf{T})$, recursively defined as follows:

- $[\![d]\!]_\theta = d$, for $d \in \mathcal{D}(\mathbf{Int})$;
- $[\![T :: d]\!]_\theta = d$ for $d \in \mathcal{D}(T_i)$;
- $[\![\mathbf{Chn} :: c]\!]_\theta = c$, for $c \in \mathcal{D}(\mathbf{Chn})$;
- $[\![x]\!]_\theta = \rho(x)$, for $x \in X$;
- $[\![trm_1 \ \boxdot_2 \ trm_2]\!]_\theta = [\![trm_1]\!]_\theta \ \boxdot_2 \ [\![trm_2]\!]_\theta$;
- $[\![\boxdot_1 \ trm]\!]_\theta = \boxdot_1 \ [\![trm]\!]_\theta$;
- $[\![len(c)]\!]_\theta = |\chi(c)|$.

Given a sequence α, let $head(\alpha)$ denote the first element of the sequence and $tail(\alpha)$ denote the sequence obtained from α by removing its first element. A satisfiability relation for an evaluation context $\theta = \langle \rho, \chi, \eta \rangle$ and a ground guard ϕ, written $\theta \models \phi$, is recursively defined as follows:

- $\theta \models True$;
- $\theta \models c[?\top]$ if $c \in C$ and $|\chi(c)| = bd(c)$;
- $\theta \models c[?\bot]$ if $c \in C$ and $|\chi(c)| = 0$;
- $\theta \models c[?\langle trm_1, \ldots, trm_k \rangle]$ if $c \in C$, $|\chi(c)| > 0$ and $head(\chi(c)) = \langle t_1, \ldots, t_k \rangle$, where, for $i \in [1, \ldots .k]$, $t_i = [\![trm_i]\!]_\theta$, if $trm_i \in trm$, and $t_i = (head(\chi(c)))_i$, if $trm_i = \_$;
- $\theta \models trm_1 \ \odot \ trm_2$, if $\odot \in \{\leq, \geq\}$, $type(trm_1) = type(trm_2) = \mathbf{Int}$ and $[\![trm_1]\!]_\theta \ \odot \ [\![trm_2]\!]_\theta$;
- $\theta \models trm_1 = trm_2$ if $type(trm_1) = type(trm_2) \in \mathbf{T}$ and $[\![trm_1]\!]_\theta = [\![trm_2]\!]_\theta$;
- $\theta \models \phi_1 \wedge \phi_2$ if $\theta \models \phi_1$ and $\theta \models \phi_2$;
- $\theta \models \phi_1 \vee \phi_2$ if $\theta \models \phi_1$ or $\theta \models \phi_2$;
- $\theta \models \neg\phi$ if it is not the case that $\theta \models \phi$.

Most of the cases above are obvious, with the possible exception of the guard $c[?\langle trm_1, \ldots, trm_k \rangle]$. The corresponding semantic condition simply requires that the first message in the non-empty channel c ($head(\chi(c))$) be equal to the tuple obtained by substituting each term in $\langle trm_1, \ldots, trm_k \rangle$ different from the wild card symbol _ with their valuation ($[\![trm_i]\!]_\theta$), and all the wild cards with the message component in the same position within the actual message ($(head(\chi(c)))_i$).

Given a function $f : X \to Y$ and two elements $x \in X$ and $y \in Y$, we denote with $f[x := y]$ the function f' which is equal to f on all the elements of X except for x, to which it assigns the value y. For a sequence α and $i \in [1, \ldots, |\alpha|]$, we denote with $(\alpha)_i$ the i-th projection of α, i.e., the element of α occurring in the i-th position in the sequence. We can now provide the semantics of actions by illustrating how their execution changes the evaluation context. To this end, given an evaluation context $\theta = \langle \rho, \chi, \eta \rangle$, we define $\theta \xrightarrow{a} \langle \rho', \chi', \eta' \rangle$, for each atomic action $a \in A$, as follows:

- $\theta \xrightarrow{x:=trm} \langle \rho', \chi, \eta \rangle$, where $\rho' = \rho[x := [\![trm]\!]_\theta]$;
- $\theta \xrightarrow{c[?]\langle t_1, \ldots, t_k \rangle} \langle \rho', \chi, \eta \rangle$, where $c \in C_I \cup C_E$, $|\chi(c)| > 0$ and
  - $\rho'(t_i) = (head(\chi(c)))_i$, if $t_i \in X$;
  - $\rho'(x) = \rho(x)$, for all $x \in X \setminus \{t_1, \ldots, t_k\}$;
- $\theta \xrightarrow{c?\langle t_1, \ldots, t_k \rangle} \langle \rho', \chi', \eta \rangle$, where $c \in C_I \cup C_E$, ρ' is defined as in the previous case, and
  - if $c \in C_I$ and $|\chi(c)| > 0$, then $\chi' = \chi[c := tail(\chi(c))]$;
  - $\chi' = \chi$, otherwise;
- $\theta \xrightarrow{c!\langle trm_1, \ldots, trm_k \rangle} \langle \rho', \chi', \eta' \rangle$, where
  - $c \in C_I$ implies $\eta' = \eta$ and
    - if $|\chi(c)| < bd(c)$, then $\chi' = \chi[c := \chi(c) \cdot \langle [\![trm_1]\!]_\theta, \ldots, [\![trm_k]\!]_\theta \rangle]$;
    - $\chi' = \chi$, otherwise;
  - $c \in C_E$ implies $\chi' = \chi$ and
    - if $\eta(c) = \bot$, then $\eta' = \eta[c := \langle [\![trm_1]\!]_\theta, \ldots, [\![trm_k]\!]_\theta \rangle]$;
    - $\eta' = \eta$, otherwise;

Notice that the "read with side effect" action $c?\langle t_1, \ldots, t_k\rangle$ is executable also over external channel. However, the side effect only occurs when applied on internal channels, while the content of the channel is retained when it is an external channel. Hence, when applied on external channels, this action coincides with the action $c[?]\langle t_1, \ldots, t_k\rangle$. Notice also that actions are modelled as non-blocking operations: they are always executable and, when the relevant condition is not satisfied, they act as null operations that leave the evaluation context unchanged.

The relation $\xrightarrow{a}$ can be extended inductively to sequences of atomic actions in the obvious way. Formally:

- $\theta \xrightarrow{\epsilon} \theta$;
- $\theta \xrightarrow{a;\alpha'} \theta'$, if $\theta \xrightarrow{a} \theta''$ and $\theta'' \xrightarrow{\alpha'} \theta'$.

References

[1] R. Alur, S. Kannan, M. Yannakakis, Communicating hierarchical state machines, in: Automata, Languages and Programming, in: Lect. Notes Comput. Sci., vol. 1644, Springer, Berlin, Heidelberg, 1999, pp. 169–178.

[2] D. Amalfitano, N. Amatucci, A.R. Fasolino, U. Gentile, G. Mele, R. Nardone, V. Vittorini, S. Marrone, Improving code coverage in android apps testing by exploiting patterns and automatic test case generation, in: Proceedings of the 2014 International Workshop on Long-Term Industrial Collaboration on Software Engineering, ACM, 2014, pp. 29–34.

[3] S. Anand, E.K. Burke, T.Y. Chen, J.A. Clark, M.B. Cohen, W. Grieskamp, M. Harman, M.J. Harrold, P. McMinn, An orchestrated survey of methodologies for automated software test case generation, J. Syst. Softw. 86 (2013) 1978–2001.

[4] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, M. Yamamoto, Modbat: a model-based API tester for event-driven systems, in: Proc. 9th Haifa Verification Conference, HVC 2013, Haifa, Israel, in: Lect. Notes Comput. Sci., vol. 8244, Springer, 2013, pp. 112–128.

[5] G. Barberio, B. Di Martino, N. Mazzocca, L. Velardi, A. Amato, R. De Guglielmo, U. Gentile, S. Marrone, R. Nardone, A. Peron, V. Vittorini, An interoperable testing environment for ERTMS/ETCS control systems, in: Computer Safety, Reliability, and Security, in: Lect. Notes Comput. Sci., vol. 8696, Springer International Publishing, 2014, pp. 147–156.

[6] M. Benerecetti, S. Minopoli, A. Peron, Analysis of timed recursive state machines, in: 17th International Symposium on Temporal Representation and Reasoning, TIME, IEEE, 2010, pp. 61–68.

[7] M. Benerecetti, A. Peron, Timed recursive state machines: expressiveness and complexity, Theor. Comput. Sci. 625 (2016) 85–124.

[8] S. Bernardi, F. Flammini, S. Marrone, N. Mazzocca, J. Merseguer, R. Nardone, V. Vittorini, Enabling the usage of UML in the verification of railway systems: the dam-rail approach, Reliab. Eng. Syst. Saf. 120 (2013) 112–126.

[9] D. Bjørner, New results and trends in formal techniques and tools for the development of software for transportation systems: a review, in: Proceedings 4th Symposium on Formal Methods for Railway Operation and Control Systems, FORMS03, LHarmattan Hongrie, Budapest, 2003.

[10] C. Braunstein, A.E. Haxthausen, W. Huang, F. Hübner, J. Peleska, U. Schulze, L.V. Hong, Complete model-based equivalence class testing for the ETCS ceiling speed monitor, in: Proceedings of the Formal Methods and Software Engineering, 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3–5, 2014, pp. 380–395.

[11] CENELEC, En50129, Railway Applications—Communication, Signaling and Processing Systems—Safety Related Electronic Systems for Signalling, BSI, 2003.

[12] CENELEC, EN50128 Communication, Signalling and Processing Systems—Software for Railway Control and Protection Systems, 2011.

[13] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NuSMV: a new symbolic model checker, Int. J. Softw. Tools Technol. Transf. 2 (2000) 2000.

[14] U. Gentile, S. Marrone, G. Mele, R. Nardone, A. Peron, Test specification patterns for automatic generation of test sequences, in: Formal Methods for Industrial Critical Systems, Springer International Publishing, 2014, pp. 170–184.

[15] M. Glinz, Statecharts for requirements specification—as simple as possible, as rich as needed, in: Proceedings of the ICSE 2002 International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, Orlando, Florida, USA.

[16] G. Hamon, A denotational semantics for stateflow, in: Proceedings of the 5th ACM International Conference on Embedded Software, EMSOFT'05, ACM, New York, NY, USA, 2005, pp. 164–172.

[17] G. Hamon, J. Rushby, An operational semantics for stateflow, Int. J. Softw. Tools Technol. Transf. 9 (2007) 447–456.

[18] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Program. 8 (1987) 231–274.

[19] D. Harel, A. Naamad, The statemate semantics of statecharts, ACM Trans. Softw. Eng. Methodol. 5 (1996) 293–333.

[20] A.E. Haxthausen, J. Peleska, Model checking and model-based testing in the railway domain, in: Formal Modeling and Verification of Cyber-Physical Systems, Springer, 2015, pp. 82–121.

[21] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, et al., Using formal specifications to support testing, ACM Comput. Surv. 41 (2009) 9.

[22] G. Holzmann, The Spin Model Checker: Primer and Reference Manual, first edition, Addison–Wesley Professional, 2003.

[23] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: a model transformation tool, Sci. Comput. Program. 72 (2008) 31–39.

[24] R. Lanotte, A. Maggiolo-Schettini, A. Peron, S. Tini, Dynamic hierarchical machines, Fundam. Inform. 54 (2002) 237–252.

[25] N. Leveson, M. Heimdahl, H. Hildreth, J. Reese, Requirements specification for process-control systems, IEEE Trans. Softw. Eng. 20 (1994) 684–707.

[26] A. Maggiolo-Schettini, A. Peron, S. Tini, A comparison of statecharts step semantics, Theor. Comput. Sci. 290 (2003) 465–498.

[27] S. Marrone, F. Flammini, N. Mazzocca, R. Nardone, V. Vittorini, Towards model-driven V&V assessment of railway control systems, Int. J. Softw. Tools Technol. Transf. 16 (2014) 669–683.

[28] S. Marrone, R. Nardone, A. Tedesco, P. D'Amore, V. Vittorini, R. Setola, F. De Cillis, N. Mazzocca, Vulnerability modeling and analysis for critical infrastructure protection applications, Int. J. Crit. Infrastruct. Prot. 6 (2013) 217–227.

[29] S. Marrone, R.J. Rodríguez, R. Nardone, F. Flammini, V. Vittorini, On synergies of cyber and physical security modelling in vulnerability assessment of railway systems, Comput. Electr. Eng. 47 (2015) 275–285.

[30] R. Nardone, U. Gentile, M. Benerecetti, A. Peron, V. Vittorini, S. Marrone, N. Mazzocca, Modeling railway control systems in Promela, in: Formal Techniques for Safety-Critical Systems, vol. 596, Springer International Publishing, 2016, pp. 121–136.

[31] R. Nardone, U. Gentile, A. Peron, M. Benerecetti, V. Vittorini, S. Marrone, R. De Guglielmo, N. Mazzocca, L. Velardi, Dynamic state machines for formalizing railway control system specifications, in: Formal Techniques for Safety-Critical Systems, Springer International Publishing, 2014, pp. 93–109.

[32] OMG-UML2, Unified Modeling Language: Infrastructure and Superstructure, OMG, 2011, Version 2.4.1 formal/11-08-05.

[33] J. Peleska, Industrial-strength model-based testing—state of the art and current challenges, Electron. Proc. Theor. Comput. Sci. 111 (2013) 3–28.

[34] A. Petrenko, A. da Silva Simão, J.C. Maldonado, Model-based testing of software and systems: recent advances and challenges, Int. J. Softw. Tools Technol. Transf. 14 (2012) 383–386.
[35] H. Pflügl, C. El-Salloum, I. Kundner, CRYSTAL, critical system engineering acceleration, a truly European dimension, ARTEMIS Mag. 14 (2013) 12–15.
[36] D.C. Schmidt, Guest editor's introduction: model-driven engineering, Computer 39 (2006) 25–31.
[37] UIC, ERTMS/ETCS Class1 System Requirements Specification, Ref. SUBSET-026, Issue 2.2.2 2002.
[38] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.