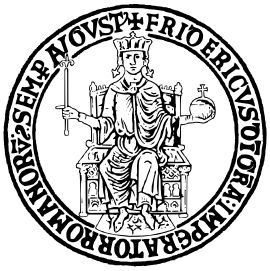
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Model-based Testing and Model Checking for Safety-critical Hierarchical Systems

Relatori
Professor Adriano Peron
Professor Massimo Benerecetti
Dottor Fabio Mogavero

Correlatore
Professor Guglielmo Tamburrini

Candidato
Luigi Libero Lucio Starace
N97/243

Anno Accademico 2017–2018

Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Informatica

Model-based Testing and Model Checking for Safety-critical Hierarchical Systems

Relatori
Professor Adriano Peron
Professor Massimo Benerecetti
Dottor Fabio Mogavero

Correlatore
Professor Guglielmo Tamburrini

Candidato
Luigi Libero Lucio Starace
N97/243

Anno Accademico 2017–2018

Abstract

The ever-growing magnitude and complexity of computer systems, along with the pressure to drastically reduce system development time and the rising society's reliance on their correctness, make the delivery of *correct* systems an extremely important and difficult task. Traditional systems verification techniques such as testing often prove themselves inadequate to face such challenges, thus, both in academia and industry, techniques are sought to reduce and ease the verification efforts while increasing their effectiveness. The model-based verification approach introduces formal methods in the verification process and allows for the application of techniques ranging from automatic test case generation to model checking. This thesis work extends previous research in model-based verification conducted within the CRYSTAL European project. In such previous research, the *Dynamic State Machines* (DSTM) formal specification language was introduced, along with an automatic test case generation procedure from a DSTM model. In this thesis work, several crucial issues with the previously-defined procedure are detected and addressed by devising a novel procedure. Furthermore, a novel logic formalism (*Hierarchical Linear-time Temporal Logic* – HLTL^{f}) is proposed and theoretical foundations for DSTM model checking are laid down.

Sommario

Le sempre crescenti estensione e complessità dei sistemi informatici, unitamente alla pressione per ridurre drasticamente i tempi di sviluppo e al sempre maggiore affidamento che la società ripone nella loro correttezza, rendono la creazione di sistemi *corretti* un compito estremamente importante e arduo. Le tecniche di verifica tradizionali come il *testing* spesso si rivelano inadeguate ad affrontare il problema e, quindi, sia in ambito accademico che industriale si ricercano nuove tecniche che possano ridurre gli sforzi di verifica e allo stesso tempo aumentare l'efficacia del processo. L'approccio alla verifica basato su modelli (*model-based*) introduce nel processo di verifica i metodi formali e rende possibile l'applicazione di diverse tecniche che spaziano dalla generazione automatica di casi di test al *model checking*. Questo lavoro di tesi estende attività di ricerca precedenti nell'ambito delle tecniche di verifica basate su modelli condotte all'interno del progetto Europeo CRYSTAL. In queste precedenti attività è stato definito il linguaggio di specifica formale *Dynamic State Machines* (DSTM) ed è stata prodotta una procedura per generare automaticamente casi di test a partire da un modello DSTM. Nel presente lavoro di tesi sono state individuate diverse criticità nella procedura precedentemente definita. Queste criticità sono state risolte proponendo una nuova procedura per la generazione automatica di casi di test. Inoltre, viene proposto un nuovo formalismo logico (*Hierarchical Linear-time Temporal Logic* – HLTL$^{\prime}$) e vengono poste le basi teoriche necessarie ad arrivare a una procedura di *model checking* per modelli DSTM.

Contents

Introduction

Systems verification: model-based testing and model checking

In today's world, computer and software systems are ubiquitous and involved in almost every aspect of daily life. From railway-traffic control systems to smartphones, from medical appliances to the stock exchange market, from power plants to communication networks, society relies on such systems to an ever-growing extent, making their reliability an issue of great social importance. Furthermore, it is increasingly more rare to find isolated computer systems, as they are typically embedded in larger contexts, interacting with several other concurrently-executing systems over wired and wireless networks. This trend, besides the fact that they perform increasingly more complicated tasks, makes computer systems' complexity grow apace, along with the difficulty of their *verification*.

A system's verification is the process of checking that a system, a design, or a prototype meets certain requirements obtained from a given *specification*. A system is deemed *correct* with regard to a specification if it fulfils all of the specification's requirements. When dealing with the verification of software systems, the major techniques used in practice are *code inspection* and *testing*. Code inspection, also known as *peer review*, consists in a careful scrutiny of the source code carried on by programmers/software engineers that preferably are not involved in the development of the system under verification. This kind of analysis is completely static and the software is not executed. The testing technique, on the contrary, is dynamic and actually executes the software under verification. A suite of *test cases*, each specifying inputs and expected system behaviour (derived from a specification), are typically produced by software testers and used to inspect a – usually very small – subset of the possible system's behaviours. Both these techniques are rather effective at detecting errors – and are in fact largely used in software projects – but have downsides that it is necessary to consider.

Firstly, as noted by Dijkstra in [1], these techniques "*can be a very effective way to show the presence of bugs, but are hopelessly inadequate for showing their absence*". Because of this, these techniques alone are not sufficient to provide the high level of correctness confidence expected for the so-called *safety-critical* systems, i.e. those systems whose malfunctioning can cause severe damage to people or to the environment. Safety-critical systems include, for example, control systems for vehicles, medical appliances, nuclear

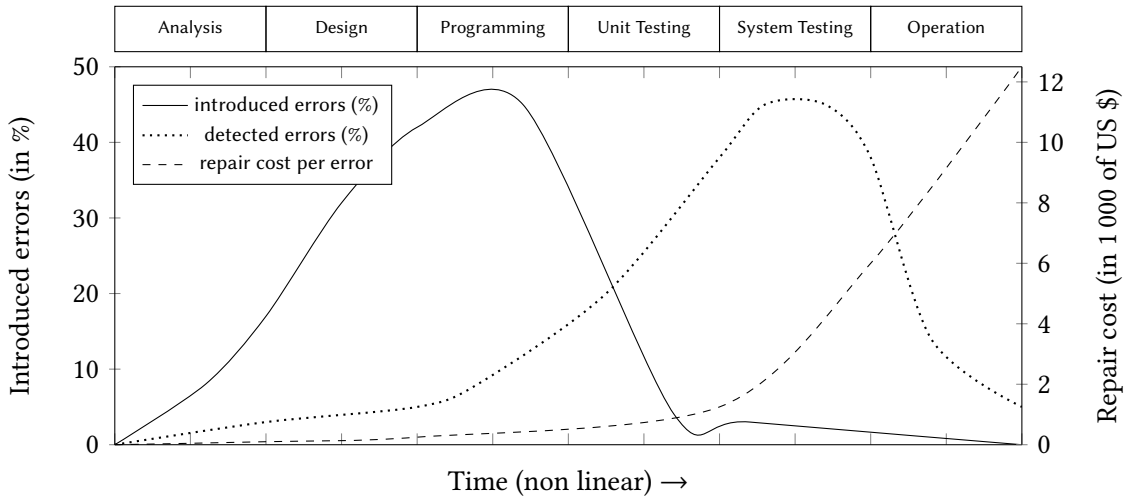


Figure 1: Software lifecycle and error introduction, detection, and repair costs [3]

power plants, etc.

Secondly, testing and code inspection are not very effective on concurrent systems, since intuition often fails to fully grasp the complexity and non-deterministic nature of concurrency. To put it in Lamport and Owicki's words [2], "*there is a rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors*" and "*the only way we can be sure that a concurrent program does what we think it does is to prove rigorously that it does it.*"

Finally, testing and code inspection can only be applied in the later stages of the software lifecycle and, as shown in Figure 1, the repair cost for errors detected in the later stages is way higher than the cost for error detected in the initial phases.

The model-based verification approach tries to overcome the previously-discussed shortcomings and to obtain more effective verification techniques by applying formal methods, i.e. the "*mathematics for modelling and analysing ICT systems*" [3]. The first essential step to the application of model-based verification techniques, as the name itself suggests, is to abstract an accurate and unambiguous mathematical representation of the system under verification, namely the *formal specification*. Often, this modelling phase alone – prior to the application any form of verification – leads to the detection of incompleteness, ambiguities, and inconsistencies in the "informal" specifications.

Once a formal specification is defined, it is possible to apply *model-based testing* techniques to automatically generate test cases, or *model checking* techniques to mathematically prove that a certain property holds in every possible behaviour of the model.

About this thesis work

This thesis work originates from research on formal modelling languages and model-based testing conducted within the context of the ARTEMIS Joint Undertaking project CRYSTAL

(CRitical sYSTem engineering AcceLeration) [4], and in particular in [5, 6, 7]. In these works, a novel modelling formalism called Dynamic STate Machines (DSTM), explicitly devised to model interrupting hierarchical systems and to meet industrial requirements in design, verification and validation of complex control systems, is introduced, along with a procedure to automatically derive test cases from a DSTM model.

Chapter 1, after giving a brief overview of DSTM key features and a comparison with other modelling formalisms, describes DSTM formal syntax and semantics. Chapter 2 proposes a new procedure to automatically generate test cases from a DSTM model, addressing several issues afflicting the original procedure described in [5, 6]. Finally, Chapter 3, in an initial effort to lay down a theoretical foundation for DSTM model checking, proposes a logic (Hierarchical Linear-time Temporal Logic with Interrupts - $\text{HLTL}^{\lightning}$) designed to predicate about computations of interrupting hierarchical systems. A concrete instantiation of this logic is then given in terms of Communicating Structured Automata with Interrupts ($\text{CSA}^{\lightning}$), which are simpler systems than DSTM, yet maintain the most important characteristics of hierarchy, concurrency and the possibility of interrupts, and therefore are better-suited for a preliminary study. After restricting the focus to the local fragment of $\text{HLTL}^{\lightning}$ ($\text{HLTL}^{\lightning}_{\text{L}}$) and to the sub-class of non-recursive $\text{CSA}^{\lightning}$ (Communicating Hierarchical Automata – $\text{CHA}^{\lightning}$), algorithmic results for the emptiness problem of $\text{CHA}^{\lightning}$, the satisfiability of $\text{HLTL}^{\lightning}_{\text{L}}$ formulae over the class of $\text{CHA}^{\lightning}$ computations, and finally model checking for $\text{CHA}^{\lightning}$ models against $\text{HLTL}^{\lightning}_{\text{L}}$ properties are provided.

–1–
Dynamic State Machines: a formal modelling language

Contents: **1.1 A comparison with other modelling languages. 1.2 DSTM Syntax.** 1.2.1 Control flow – 1.2.2 Data flow. **1.3 DSTM Semantics.** 1.3.1 Semantics of transition decorations – 1.3.2 Machine instantiation – 1.3.3 Semantics by means of a Labelled Transition System.

The Dynamic STate Machine (DSTM) formalism is a recently-developed modelling language originally proposed by Benerecetti et al. in [5] and originating within the context of the ARTEMIS Joint Undertaking project CRYSTAL (CRitical sYSTem engineering AcceLeration) [4]. DSTM is explicitly devised to meet industrial requirements in design, verification and validation of complex control systems, and includes in its formal framework both complex control flow constructs such as asynchronous forks, preemptive termination, recursive execution and complex data flow constructs such as custom complex type definition, parametric machines, and inter-process communication.

1.1 A comparison with other modelling languages

A great deal of formal languages have been proposed and used for modelling purposes. For example, Finite State Machines (FSMs) are often used to model sequential circuits and communication protocols [8]. Statecharts, introduced by Harel in [9], are widely used in software engineering and extend FSMs with a notion of hierarchy and concurrency. Both in the UML [10] and in the STATEMATE [11] variants, statecharts have no concept of modules and instances. If multiple instances of the same module are required, it is necessary to define each one explicitly. Moreover, dynamic instantiation is not allowed. Communicating Hierarchical Machines (CHMs), proposed by Alur et al. in [12], introduce the notion of model as a sequence of modules, each made by nodes and boxes containing sequences of modules. A transition entering a box is a transition instantiating all the modules associated with the box, thus allowing for more succinct representations than

statecharts. The introduction of modules also allows for the definition of Recursive State Machines (RSMs) [13] in which a module can instantiate itself but concurrency is not allowed.

DSTMs borrow many syntactic elements from UML statecharts, and extend them with the notion of module and with the possibility of recursion and dynamic instantiation (which are not allowed in CHMs). Notice that a machine is capable of dynamically instantiating one of its modules when the number of concurrently-executing instances of said module is decided at run-time.

1.2 DSTM Syntax

This section formally defines DSTM syntax, with Subsection 1.2.1 detailing the syntax of control flow elements and Subsection 1.2.2 defining the syntax of data flow elements.

1.2.1 Control flow

Given a set P of parameters, a Dynamic STate Machine (DSTM) model is a sequence of machines $M_1, M_2, \ldots, M_n$ communicating over a set X of global variables and a set C of global communication channels. Machine M_1 is the *initial machine*, namely the highest level of the hierarchical system, and cannot be parametric. Machines M_i, $i \in \{2, \ldots, n\}$, are possibly parametric over a set of parameters $P_i \subseteq P$. Parameters are aliases for channels and variables and are actualized at runtime when instantiating a parametric machine, allowing multiple instantiations of the same machine with different parameter values. When a parametric machine is instantiated, each parameter is mapped to its actual value by means of a *parameter-substitution function*. A (possibly parametric) machine M_i represents a module in the specification and is defined as a state-transition diagram composed by vertices and transitions connecting them. The following kinds of vertices are possible:

node basic control state of a machine;

entering node initial pseudo-node of a machine. A machine may specify multiple entering nodes, corresponding to different initial conditions;

initial node default entering pseudo-node of a machine, to be used when no entering node is explicitly specified. There must be exactly one for each machine;

exit node final (or exiting) node of a machine. A machine may specify multiple exiting nodes, corresponding to different termination conditions;

box node modelling the parallel activation of machines associated with the box itself. A transition entering a box represents the parallel activation of the corresponding machines, while a transition exiting a box corresponds to a return from said activation.

fork control pseudo-node modelling the activation of new processes. Such activation may be either *synchronous* (the forking process is suspended and waits for the activated processes to terminate) or *asynchronous* (the forking process continues its activity along the newly-activated processes).

join control pseudo-node used to synchronize the termination of concurrently executing processes or to force their termination when necessary (*preemptive* join).

In the description above the vertices corresponding to stable, meaningful control points are called *nodes*, as opposed to *pseudo-nodes*, which are only transient points.

Transitions represent changes in the control state of a machine. A transition is labelled with a name and decorated with a *trigger* (an input event originating from the external environment or from other concurrent machines), a *guard* (a Boolean condition on the current contents of variables and channels) and an *action* (one or more statements on variables and channels). For a transition to be fired it is necessary that its trigger is fulfilled and that its guard is satisfied. When a transition fires, its action is executed with possible side-effects.

For the sake of exposition, assume the availability of the syntactic categories of well-formed triggers Ξ_P, guards Φ_P, actions $\mathcal{A}_P$, and parameter-substitution functions Υ_P over a set P of parameters. These syntactic categories will be formally defined in Subsection 1.2.2. Let τ denote the trivial trigger (no external event is required), *True* denote the trivial guard (always satisfied), and ε denote the empty action (no side effects).

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_i is a machine over X, C and P of the form

$$\langle P_i, N_i, En_i, df_i, Ex_i, Bx_i, Y_i, Fk_i, \mathcal{J}n_i, \Lambda_i\rangle, \text{ where:}$$

  - $P_i \subseteq P$ is the local set of parameters of the i-th machine. If $i = 1$, $P_i = \varnothing$ (the initial machine cannot be parametric).
  - N_i is a finite set of nodes;
  - En_i is a finite non-empty set of entering pseudo-nodes;
  - $df_i \in En_i$ is the default entering pseudo-node;
  - $Ex_i \subseteq N_i$ is the set exiting nodes;
  - Bx_i is a finite set of boxes;
  - $Y_i : Bx_i \to \{1, \ldots, n\}^*$ is a function mapping each box to a sequence of machine indices;
  - Fk_i (resp. $\mathcal{J}n_i$) is the finite set of fork (resp. join) pseudo-nodes.

– Λ_i defines the machine's transitions and is a tuple of the form

$$\langle T_i, Src_i, Trg_i, Dec_i, Inst_i \rangle, \text{ where:}$$

  * T_i is a finite set of transition labels;
  * $Src_i : T_i \rightarrow Source_i$ is a function associating each transition label with a source vertex. All vertices except for final nodes can be source of a transition. Formally, $Source_i = (N_i \setminus Ex_i) \cup En_i \cup Bx_i \cup (Bx_i \times Ex(D)) \cup Fk_i \cup (Fk_i \times \{\downarrow\}) \cup \mathcal{J}n_i$, with $Ex(D) = \bigcup_{j=1}^{n} Ex_j$;
  * $Trg_i : T_i \rightarrow Target_i$ is a function associating each transition label with a target vertex. Entering pseudo-nodes cannot be target of any transition. $Target_i = N_i \cup Bx_i \cup (Bx_i \times En(D)) \cup Fk_i \cup \mathcal{J}n_i \cup (\mathcal{J}n_i \times \{\otimes\})$, with $En(D) = \bigcup_{j=1}^{n} Ex_j$;
  * $Dec_i : T_i \rightarrow \Xi_{P_i} \times \Phi_{P_i} \times \mathcal{A}_{P_i}$ is a function associating each transition label with its decoration.
  * $Inst_i : T_i \rightharpoonup \Upsilon_P$ is a partial function associating transition labels whose target is a box with a sequence of parameter-substitution functions, representing the parameter actualization for each machine instantiated by the box.

Note that the pairing of a fork pseudo-node with the symbol $\downarrow$ (resp. of a join pseudo-node with the symbol $\otimes$) is used to qualify a source fork as asynchronous (resp. a target join as preemptive).

Example 1 (The *Counting* DSTM – part 1)**.** As an example, consider the DSTM $Counting = \langle M_1, M_2, M_3, X, C, P \rangle$ where M_1, M_2, M_3 are respectively the machines *Main*, *Counter* and *Incrementer* represented in Figure 1.1. In the proposed graphical formalism, default entering pseudo-nodes are depicted as black circles, entering pseudo-nodes as white circles, final nodes as crossed-out white circles. Boxes are represented by rectangles and decorated with a comma-separated list of associated machines enclosed in square brackets. Nodes are drawn as rounded rectangles and fork and join pseudo-nodes are represented by black bars. Each node and pseudo-node is decorated with its name. Transitions are, as usual, drawn as directed edges between the source and the target vertices. Formally, we have that machines M_1 (*Main*), M_2 (*Counter*) and M_3 (*Incrementer*) are respectively defined as follows:

- $P_1 = \emptyset$; $En_1 = df_1 = \{\mathsf{initial}\}$; $Ex_1 = \{\mathsf{interrupted}, \mathsf{stopped}\}$; $N_1 = \{\mathsf{idle1}\} \cup Ex_1$; $Bx_1 = \{\mathsf{CounterBox}\}$; $Fk_1 = \emptyset$; $\mathcal{J}n_1 = \emptyset$; $Y_1 = \{(\mathsf{CounterBox}, 2)\}$;

- $P_2 = \{\mathtt{P\_to}\}$; $En_2 = df_2 = \{\mathsf{default}\}$; $Ex_2 = \{\mathsf{limit}\}$; $N_2 = \{\mathsf{idle2}\} \cup Ex_2$; $\mathcal{J}n_2 = \{\mathsf{jn}\}$; $Bx_2 = \{\mathsf{BoxIncr1}, \mathsf{BoxIncr2}\}$; $Y_2 = \{(\mathsf{BoxIncr1}, 3), (\mathsf{BoxIncr2}, 3)\}$; $Fk_2 = \{\mathsf{fk}\}$;

- $P_3 = \{\mathtt{P\_limit}\}$; $df_3 = \{\mathsf{byOne}\}$; $En_3 = \{\mathsf{byTwo}\} \cup df_3$; $Ex_3 = \{\mathsf{finished}\}$; $N_3 = \{\mathsf{SimpleIncr}, \mathsf{DoubleIncr}\} \cup Ex_3$; $Bx_3 = \emptyset$; $Fk_3 = \emptyset$; $\mathcal{J}n_3 = \emptyset$; $Y_3 = \emptyset$;

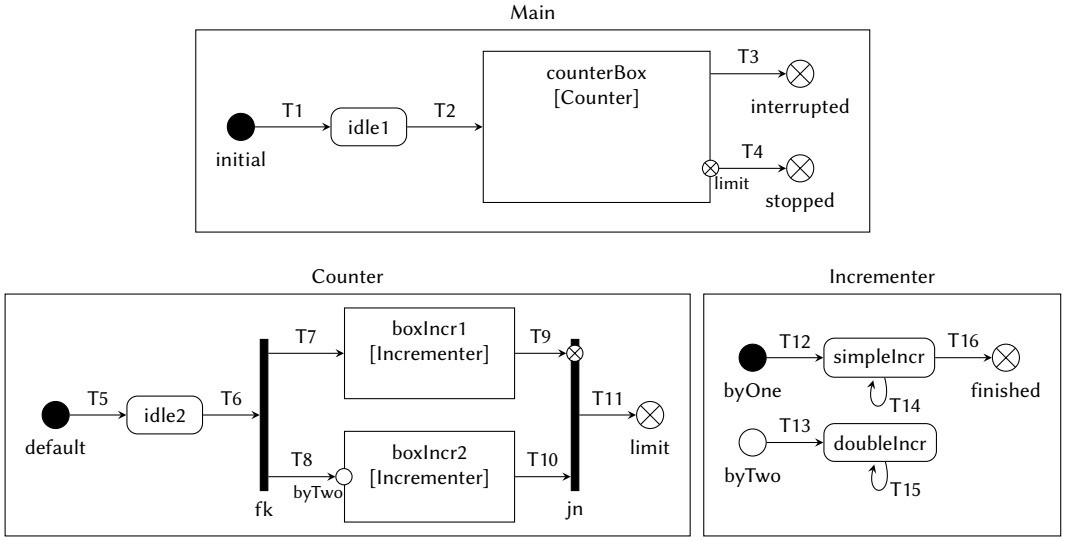


Figure 1.1: The *Counting* DSTM specification

T_1	Src_1	Trg_1	Dec_1	$Inst_1$
T1	initial	idle1	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T2	idle1	counterBox	$\langle \tau, True, \varepsilon \rangle$	`P_to=100`
T3	counterBox	interrupted	$\langle$`signal?`$, True, \varepsilon \rangle$	$\varnothing$
T4	(counterBox, limit)	stopped	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T_2	Src_2	Trg_2	Dec_2	$Inst_2$
T5	default	idle2	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T6	idle2	fk	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T7	fk	boxIncr1	$\langle \tau, True, \varepsilon \rangle$	`P_limit=P_to`
T8	fk	(boxIncr2, byTwo)	$\langle \tau, True, \varepsilon \rangle$	`P_limit=P_to`
T9	boxIncr1	(jn, $\otimes$)	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T10	boxIncr2	jn	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T11	jn	limit	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T_3	Src_3	Trg_3	Dec_3	$Inst_3$
T12	byOne	simpleIncr	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T13	byTwo	doubleIncr	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T14	simpleIncr	simpleIncr	$\langle \tau,$ `x<P_limit`$, $`x++`$\rangle$	$\varnothing$
T15	doubleIncr	doubleIncr	$\langle \tau, True,$ `x+=2`$\rangle$	$\varnothing$
T16	simpleIncr	finished	$\langle \tau,$ `x`$\geq$`P_limit`$, \varepsilon \rangle$	$\varnothing$

Table 1.1: Transition structure for the DSTM model *Counting*

Transitions, namely the structures $\Lambda_{i\in\{1,2,3\}}$, are detailed in Table 1.1. More considerations on this specification's transitions will follow in Example 2.

DSTM syntax requires for each transition $t \in T_i$ of a given machine M_i of a DSTM specification D to belong to one of the classes detailed in Table 1.2, where $\alpha \in \mathcal{A}_P$ is used to denote a generic action, $\phi \in \Phi_P$ stands for a generic guard, $\xi \in \Xi_P$ represents a generic trigger and $\hat{\xi} \in \Xi_P \setminus \{\tau\}$ denotes a non-trivial trigger. τ, *True* and ε denote, as already said, the trivial trigger, the trivial guard and the empty action, respectively.

Definition 2 (Well-formed DSTM)**.** A DSTM specification

$$D = \langle M_1, M_2, \ldots, M_n, X, C, P\rangle$$

is *well-formed* if each transition in each of its machines belongs to one of the classes defined in Table 1.2 and the following additional constraints are satisfied by each machine:

(i) only transitions whose target is a box may specify parameter-substitution functions. If $t \in T_i$ is a transition entering a box $B \in Bx_i$ such that $Y_i(B) = m_1, \ldots, m_k$ then $Inst_i(t) = \ell_1, \ldots, \ell_k$ with $\ell_j \in \Upsilon_{P_{m_j}}$ defined on the parameters of the m_j-th machine for each $j \in \{1, \ldots, k\}$;

(ii) *call by entering* transitions must have as target a pair of the form (b, en) where $b \in Bx_i$ is a box instantiating exactly one machine and $en \in En_j$ with $j = Y_i(b)$. Similarly, *return by exiting* transitions must have as source a pair of the form (b, ex) where $b \in Bx_i$ is a box instantiating exactly one machine and $ex \in Ex_j$ with $j = Y_i(b)$.

(iii) if a box $b \in Bx_i$ is the target of either a *call by entering* or a *call by default* transition having a fork pseudo-node as source, then there is no other transition whose target is b and each transition having b as source has a join pseudo-node as target.

(iv) for each join pseudo-node $jn \in \mathcal{J}n_i$ there is a single corresponding fork pseudo-node $fk \in Fk_i$. If there is an *asynchronous fork* transition exiting from fk then there is an *entering join* transition whose target is jn. Moreover, for each $jn \in \mathcal{J}n_i$ there is at most one transition $t \in T_i$ such that $Trg(t) = (jn, \otimes)$. If a box $b \in Bx_i$ is target of either a *call by default* or a *call by entering* transition whose source is fk, then b either has no exiting transitions or has *return* (either *by default*, *by exiting* or *by interrupt*) transitions whose target is jn.

In Definition 2, constraint (i) guarantees the coherence of parameter substitution functions, (ii) guarantees that the entering (resp. exiting) nodes associated with a box are actually entering (resp. exiting) nodes of the machine instantiated by said box, (iii) ensures that a box called after a fork cannot be called in other ways and has exiting transitions leading only to join pseudo-nodes, (iv) requires that for each *asynchronous fork* transition there is an *entering join* transition and that a box called after a fork fk either does not terminate at all or participates with return transitions to all the join pseudo-nodes

Transition class	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	$\langle \xi, \phi, \alpha \rangle$
asynch. fork	$Fk_i \times \{\downarrow\}$	N_i	$\langle \tau, True, \alpha \rangle$
entering join	N_i	$\mathcal{J}n_i \cup \mathcal{J}n_i \times \{\otimes\}$	$\langle \xi, \phi, \varepsilon \rangle$
exiting join	$\mathcal{J}n_i$	N_i	$\langle \tau, True, \alpha \rangle$
return by default	Bx_i	$\mathcal{J}n_i \cup \mathcal{J}n_i \times \{\otimes\}$	$\langle \tau, True, \varepsilon \rangle$
	Bx_i	$N_i \cup Bx_i \cup Fk_i$	$\langle \tau, True, \alpha \rangle$
return by exiting	$Bx_i \times Ex(D)$	$\mathcal{J}n_i \cup \mathcal{J}n_i \times \{\otimes\}$	$\langle \tau, True, \varepsilon \rangle$
	$Bx_i \times Ex(D)$	$N_i \cup Bx_i \cup Fk_i$	$\langle \tau, True, \alpha \rangle$
return by interrupt	Bx_i	$\mathcal{J}n_i \times \{\otimes\}$	$\langle \hat{\xi}, True, \varepsilon \rangle$
	Bx_i	$N_i \cup Bx_i \cup Fk_i$	$\langle \hat{\xi}, True, \alpha \rangle$
call by entering	$N_i \cup Bx_i$	$Bx_i \times En_i$	$\langle \xi, \phi, \alpha \rangle$
	$Fk_i \cup \mathcal{J}n_i$	$Bx_i \times En_i$	$\langle \tau, True, \alpha \rangle$
call by default	$N_i \cup Bx_i$	Bx_i	$\langle \xi, \phi, \alpha \rangle$
	$Fk_i \cup \mathcal{J}n_i$	Bx_i	$\langle \tau, True, \alpha \rangle$

Table 1.2: Syntactic constraints on DSTM transitions

associated with *fk*. Constraints (iii) and (iv) ensure that, at any instant in time, the control state of the machine can be located in at most one node and enforce a correspondence between join and fork pseudo-nodes. Note that this correspondence is total (each join pseudo-node has a corresponding fork) but might not be injective since there might be forks with no corresponding joins as synchronization is not mandatory.

Example 2 (The *Counting* DSTM specification – part 2)**.** Let us continue with Example 1, referring to the *Counting* DSTM represented in Figure 1.1 whose transitions are detailed in Table 1.1. Note that: transitions T1, T5, T12, T13 belong to the *implicit* transition class; T14 and T15, T16 are *internal* transitions; transitions T6 and T11 are, respectively, *entering fork* and *exiting join* transitions; T2 and T7 are *call by default* transitions while T8 is a *call by entering*. T2, T7 and T8 are transitions with a non-empty substitution function since they enter boxes instantiating parametric machines. Moreover, T8 satisfies constraint (ii) of definition 2 since its target box boxIncr2 instantiates only the *Incrementer* machine and byTwo is indeed an entering state of the latter machine. T9 and T10 are *return by default*, with the first having also the quality of being preemptive. T3, with its non-trivial trigger `signal?`, is a *return by interrupt* while T4 is a well-formed *return by exiting* since its source is (counterBox, limit) and counterBox instantiates exactly one *Counter* machine and limit is an exiting state of such instantiated machine. It is immediately ensured that *Counting* satisfies constraints (iii) and (iv) since forks and joins are only used in the *Incrementer* machine and boxes boxIncr1 and boxIncr2 have only one entering *call* transition each

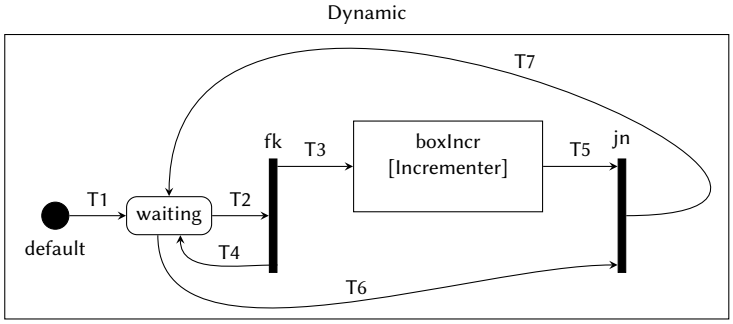


Figure 1.2: The *Dynamic* DSTM specification

T_1	Src_1	Trg_1	Dec_1	$Inst_1$
T1	default	waiting	$\langle \tau, \mathit{True}, \varepsilon \rangle$	$\varnothing$
T2	waiting	fk	$\langle$`req?`, *True*, `req?`$\langle$`V_limit`$\rangle\rangle$	$\varnothing$
T3	fk	boxIncr	$\langle \tau, \mathit{True}, \varepsilon \rangle$	`P_limit=V_limit`
T4	(fk, $\downarrow$)	waiting	$\langle \tau, \mathit{True}, \varepsilon \rangle$	$\varnothing$
T5	boxIncr	jn	$\langle \tau, \mathit{True}, \varepsilon \rangle$	$\varnothing$
T6	waiting	jn	$\langle \tau, \mathit{True}, \varepsilon \rangle$	$\varnothing$
T7	jn	waiting	$\langle \tau, \mathit{True},$ `served++`$\rangle$	$\varnothing$

Table 1.3: Transition structure for the *Dynamic* machine

and all their exiting transitions participate in the same join. By Definition 2, *Counting* is therefore a well-formed DSTM specification.

Example 3 (The *Dynamic* DSTM specification)**.** To exemplify the dynamic instantiation capabilities of DSTM and *asynchronous fork* transitions, consider the DSTM specification $\mathit{Dynamic} = \langle M_1, M_2, X, C, P \rangle$, where M_1 is the *Dynamic* machine detailed in Figure 1.2 and in Table 1.3 and M_2 is *Incrementer* machine already detailed throughout examples 1 and 2.

Transition T4 is an *asynchronous fork*, T2 is triggered by the reception of any message on the channel `req` and T3 enters boxIncr instantiating an *Incrementer* machine. T6 is an *entering join* transition and is necessary to comply with constraint (iv) of Definition 2, since there is an *asynchronous fork* transition.

Notice that the *Dynamic* DSTM is able to instantiate an unbounded number of concurrent *Incrementer* machines.

1.2.2 Data flow

This subsection will define data flow syntax for DSTMs. It starts by formalizing types, channels, variables and terms and continues with definitions for the syntactic categories of well-formed triggers Ξ_P, guards Φ_P, actions $\mathcal{A}_P$, and parameter-substitution functions Υ_P over a set P of parameters.

Types Types in DSTMs can either be *basic types*, *compound types* or *multi-types*. The set of *basic types* $\mathtt{BT} = \{\mathtt{Int}, \mathtt{Chn}, BT_1, \ldots, BT_k\}$ provides an `Int` type for integers, a `Chn` type for channel names and a set of user-defined enumeration types $BT_1, \ldots, BT_k$. The domain $\mathcal{D}(\mathtt{Int})$ is the set of integers $\mathbb{Z}$, $\mathcal{D}(\mathtt{Chn}) = \{c_1, \ldots, c_h\}$ is a set of channel names and for enumeration types $\mathcal{D}(BT_i)$ is a set of labels $\{\ell_1^i, \ldots, \ell_{s_i}^i\}$. $\mathcal{D}(\mathtt{BT})$ denotes the union of each basic type's domain $\bigcup_{b \in \mathtt{BT}} \mathcal{D}(b)$. For each basic type it is assumed that a default value in its domain is provided by means of a $default : \mathtt{BT} \to \mathcal{D}(\mathtt{BT})$ function.

Compound types are defined as tuples of *basic types*. E.g. the compound type $\mathtt{CT} = \langle BT_{j_1}, \ldots, BT_{j_k} \rangle$ is a tuple of basic types with $BT_{j_i} \in \mathtt{BT}$ and its domain is $\mathcal{D}(\mathtt{CT}) = \left\{ \langle d_1, \ldots, d_k \rangle \mid d_i \in \mathcal{D}(BT_{j_i}),\ i \in \{1, \ldots, k\} \right\}$. The class of *simple types* contains both *basic types* and *compound types*.

A *multi-type* `MT` is defined as composition of *simple types*: $\mathtt{MT} = \{ST_1, \ldots, ST_k\}$. Its domain $\mathcal{D}(\mathtt{MT})$ is defined as the union of each simple type's domain $\bigcup_{i=1}^{k} \mathcal{D}(ST_i)$. `T` is the set of all types.

Channels Each channel name $c \in \mathcal{D}(\mathtt{Chn})$ is associated with a *concrete channel* $\hat{c}$. The set of concrete channels is denoted by $\hat{C}$. Channels allow for communication with the external environment and between internal components via bounded *first-in first-out* buffers and the function $bd : C \to \mathbb{N}$ maps each channel name to the associated buffer's length. Each concrete channel conveys messages of a given type and is associated with such type by means of a function $type : \hat{C} \to \mathtt{T}$. The domain of the contents of a concrete channel $\hat{c}$ is the set of all sequences of elements in $\mathcal{D}(type(\hat{c}))$ having length at most $bd(c)$. In symbols, $\mathcal{D}(\hat{c}) = (\mathcal{D}(type(\hat{c})))^{bd(c)}$.

For the sake of clarity and readability, the set of *decorated basic types* $\mathtt{BT}^+$, defined as $\mathtt{BT}^+ = \mathtt{BT} \cup \{\mathtt{Chn[}t\mathtt{]} \mid t \in \mathtt{T}\}$, is introduced. By saying that a channel name c has decorated type `Chn[t]` it is possible to keep track of the fact that $type(\hat{c}) = t \in \mathtt{T}$. The domain of a decorated channel type is defined as $\mathcal{D}(\mathtt{Chn[t]}) = \{c \in C \mid type(\hat{c}) = \mathtt{t}\}$.

Furthermore, channels are partitioned in *internal* and *external* channels. Internal channels, whose names belong to $C_I \subseteq C$, are used for communications between components and are restricted to simple types, whereas external ones, whose names belong to $C_E \subseteq C$, are used for communications with the external environment and are restricted to having bounded buffers of length 1. As said, internal and external channels form a partition of C ($C = C_I \cup C_E$, $C_I \cap C_E = \varnothing$).

Variables Let X and P be, respectively, the sets of global variables and parameters. Each variable and parameter is associated with a decorated basic type by means of a function $type : X \cup P \to \mathtt{BT}^+$. Let $v \in X \cup P$, its domain is defined as the domain of its decorated basic type. In symbols $\mathcal{D}(v) = \mathcal{D}(type(v))$.

Terms The set of *terms* over a set of variables X and a set of parameters P, denoted by Trm_P, is defined as

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

where $x \in X$, $p \in P$, $d \in \mathcal{D}(\texttt{Int}) = \mathbb{Z}$, $c \in \mathcal{D}(\texttt{Chn}) = C$, $\ell \in \mathcal{D}(T_i)$, $\boxdot_1$ is an unary operator from set Op_1 (containing for example the classic self-increment and self-decrement operators ++ and −−) and $\boxdot_2$ is a binary operator from a set Op_2 (containing for example the classic arithmetic binary operators +,−,∗,/). For the sake of simplicity, assume that both binary and unary operators admit only integer operands. The term $len(c)$ denotes the current length of the buffer associated with $\hat{c}$.

To preserve term coherence when using unary and binary operators, each term is assigned a type by a *Type* function, defined as follows:

- $Type(v) = type(v)$ if $v \in X \cup P$;
- $Type(d) = Type(len(c)) = \texttt{Int}$ $Type(\texttt{Chn} :: c) = \texttt{Chn}$, $Type(T_i :: \ell) = T_i$;
- $Type(\boxdot_1 trm_1) = Type(trm_1 \boxdot_2 trm_2) = \texttt{Int}$ if $Type(trm_1) = Type(trm_2) = \texttt{Int}$, undefined otherwise.

A term is *well-typed* if its *Type* is defined, and *non-admissible* otherwise. A term is *ground* when it contains no occurrences of parameters.

Example 4 (On types, channels, variables and terms)**.** Consider the DSTM model $D = \langle M_1, \ldots, M_n, X, C, P \rangle$, with $X = \{x, y\}$, $C = \{c_1, c_2, c_3\}$, $P = \{p_1, p_2\}$. Let

$$\texttt{BT}_D = \{\texttt{Int}, \texttt{Chn}, \texttt{Colour}, \texttt{Shape}\}$$

be the set of *basic types* in D, with Colour and Shape user-defined enumerations with domains $\mathcal{D}(\texttt{Colour}) = \{white, red, green\}$ and $\mathcal{D}(\texttt{Shape}) = \{circle, square\}$ respectively. $\texttt{ColShape} = \langle \texttt{Colour}, \texttt{Shape} \rangle$ is a compound type whose domain $\mathcal{D}(\texttt{ColShape}) = \mathcal{D}(\texttt{Colour}) \times \mathcal{D}(\texttt{Shape})$. $\texttt{IntOrCS} = \{\texttt{Int}, \texttt{ColShape}\}$ is a multi-type whose domain is $\mathcal{D}(\texttt{IntOrCS}) = \mathcal{D}(\texttt{Int}) \cup \mathcal{D}(\texttt{ColShape})$. Let $type(\hat{c_1}) = \texttt{Int}$, $type(\hat{c_2}) = \texttt{ColShape}$ and $type(\hat{c_3}) = \texttt{IntOrCS}$, namely the decorated types of c_1, c_2 and c_3 are respectively `Chn[Int]`, `Chn[ColShape]` and `Chn[IntOrCS]`. Let $type(p_1) = \texttt{Chn}$ and $type(p_2) = \texttt{Int}$, $type(x) = \texttt{Int}$, $type(y) = \texttt{Colour}$, $type(z) = \texttt{Shape}$.

The following are well-formed terms: $x{+}y{-}12$, $len(c_2)$++, Chn::c_1, Colour::*red*. On the contrary, Shape::*circle*+$len(c_2)$ is an example of non-admissible term.

Actions An action is defined as a possibly empty sequence of atomic actions. The set of atomic actions Act_P over parameters in P is defined as

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

where $x \in X$, $type(x) = Type(trm)$, $\gamma \in P \cup C$, $Type(\gamma) = \texttt{Chn}$, $\eta_i \in X \cup \{\_\}$, $trm, trm_i \in Trm_P$, $type(\hat{\gamma}) = \langle Type(trm_1), \ldots, Type(trm_{k_\gamma}) \rangle$. To clarify the definition above, an atomic action can either be

- an assignment $x := trm$ of a value of a term trm to a variable $x \in X$. It is required that both the term and the variable have the same type. Also note that it is not possible to assign terms to parameters;

- the sending of a message over a channel γ, in symbols $\gamma!\langle trm_1, \ldots, trm_{k_\gamma}\rangle$;
- the reading (and the subsequent removal) of a message from channel γ, in symbols $\gamma?\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$. Each element in the $\eta_1, \ldots, \eta_{k_\gamma}$ tuple is either a variable to whom the corresponding element in the message will be assigned, or a *don't care* symbol '_' meaning that the corresponding element in the message is to be discarded;
- the reading (without altering the channel's content) of a message from channel γ, in symbols $\gamma[?]\langle \eta_1, \ldots, \eta_{k_\gamma}\rangle$. The $\eta_1, \ldots, \eta_{k_\gamma}$ tuple has the same meaning as in the reading with removal action.

The set $\mathcal{A}_P$ of actions over parameters in P is defined as

$$\alpha ::= \varepsilon \mid a;\alpha \qquad \text{with } a \in Act_P.$$

Triggers A trigger over the parameters in P is a Boolean expression constructed from a set of events, namely the presence of signals (messages) on a given channel. The set Ξ_P of triggers over P is defined as

$$\xi ::= \tau \mid \gamma? \mid \gamma?T \mid \xi \wedge \xi \mid \xi \vee \xi \mid \neg\xi\ ,$$

where $\gamma \in C \cup P$, τ is the always satisfied trivial trigger, $\gamma?$ is satisfied when there is a message on channel γ and $\gamma?T$ is satisfied when there is a message of type T on channel γ (useful with channels having a *multi-type* type).

Guards A guard over a set of parameters P is a Boolean expression constructed from a atomic guards by means of Boolean connectives. An atomic guard can be either

- the trivial, always satisfied, guard *True*;
- of the form $\gamma[?\top]$ or $\gamma[?\bot]$, checking that the buffer associated with channel γ is respectively full or empty;
- of the form $\gamma\left[?\langle trm_1, \ldots, trm_{k_\gamma}\rangle\right]$, with $trm_i \in Trm_P \cup \{\_\}$, checking for the component-by-component equivalence of the first message stored in the channel's buffer with the given term tuple. As with reading actions, a *don't care* symbol '_' in the i-th position is used to specify that the i-th component of the first message is to be ignored in the checking operation.
- a comparison of two terms of the form $trm_1 \odot trm_2$, testing for equality or inequality. When using inequality operators, both terms must have `Int` *Type*.

Formally, the set Φ_P of all guards over P is defined as

$$\phi ::= True \mid \gamma[?\top] \mid \gamma[?\bot] \mid \gamma\left[?\langle trm_1, \ldots, trm_{k_\gamma}\rangle\right] \mid trm_1 \odot trm_2 \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi,$$

where $\gamma \in C \cup P$, $type(\gamma) = \texttt{Chn}$, $type(\hat{\gamma}) = \langle Type(trm_1), \ldots, Type(trm_{k_\gamma})\rangle$, $trm_i \in Trm_P$, $\odot \in \{\leq, \geq, =\}$.

Parameter-substitution functions A parameter-substitution function over the set of parameters $\overline{P} \subseteq P$ is a partial function $subst : P \rightharpoonup Trm_P$. The set of all parameter-substitution functions over $\overline{P}$ is denoted by $\Upsilon_{\overline{P}}$.

Example 5 (On actions, triggers and guards)**.** Consider the DSTM *Data* given in Example 4. $x := 1$, $x := len(c_1) + 2$, $y :=$ `Colour::`*red* are all valid assignment actions. $c_1!\langle 99\rangle$, $c_2!\langle y,$ `Shape::`*square*$\rangle$, $c_3!\langle x\rangle$, $c_3!\langle y, z\rangle$ are valid channel writing actions. $c_2?\langle y, \_\rangle$ is the action reading (and discarding) the first message on channel c_2 and assigning the first `Colour` component of said message (whose type is the compound type `ColShape`) to the y variable. Note that $c_2?\langle y\rangle$ would not have been a valid action due to type mismatch. $c_1?\langle \_\rangle$ has the only side-effect of removing the first message from channel c_1, while $c_1[?]\langle x\rangle$ has no side-effect on c_1 and assigns to x the value of the first **`Int`** message in the channel.

$c_1? \vee c_2?$ is the trigger firing when there is any message on channels c_1 or c_2. $c_3?$**`Int`** fires only when there is a message of type **`Int`** on channel c_3. $c_3?$**`Int`** $\vee$ $c_3?$`ColShape` would be equivalent to $c_3?$, since multi-type `IntOrCS` is defined as the union of **`Int`** and `ColShape`.

Guard $c_1[?\bot] \wedge c_2[?\top]$ is satisfied when c_1's buffer is empty and c_2's buffer is full. $x + 1 \leq 10$ is a valid guard, as is `Colour::`*red* $= y$. $c_2[?\langle$`Colour::`*green*$, \_\rangle]$ is satisfied when the first message on channel c_2 as its first component equal to `Colour::`*green*.

1.3 DSTM Semantics

The evolution of a DSTM consists in a sequence of instantaneous reactions called *steps*. A step is a maximal set of transitions that are triggered by the current system state and by the state of the external channels.

This section begins with a definition of the *evaluation context* relative to which the semantics of ground terms, actions, triggers and guards is defined.

As already mentioned, DSTM allows for parametric machines whose parameters are actualized at execution time whenever a parametric machine is instantiated. Parameters do not hold any value during execution and serve only as placeholders to be substituted with the actual value determined by the parameter-substitution function. Therefore, DSTM semantics is defined over *ground machines*, namely machines in which actions, triggers and guards contain no parameters. Ground machines are obtained from parametric machines by applying the appropriate parameter-substitution functions, as detailed in Subsection 1.3.2.

With the previous elements in place, Subsection 1.3.3 provides formal semantics for DSTM models by means of an LTS (Labelled Transition System) describing its behaviour.

1.3.1 Semantics of transition decorations

Definition 3 (DSTM evaluation context)**.** An evaluation context θ is a tuple $\langle \rho, \chi, \eta\rangle$ where:

- $\rho : X \to \mathcal{D}(\mathtt{BT})$ is the evaluation function for variables;
- $\chi : C_I \cup C_E \to (\mathcal{D}(\mathtt{T}))^* \cup \{\bot\}$ is the evaluation function for channels relative to the current step. Each channel is associated with a sequence of messages or with $\bot$ if empty;
- $\eta : C_E \to (\mathcal{D}(\mathtt{T}) \cup \{\bot\})$ is the evaluation function for external channels with regard to the next step. Each channel is associated with a single message (as said in 1.2.2, external channels must have buffers of length 1) or with $\bot$ if there is no message.

The set of all evaluation contexts is denoted by Θ.

In order to define the semantics of terms, actions, triggers and guards, with respect to a certain evaluation context θ, it is necessary to define:

- a term *denotation* function assigning to each ground term *trm* its evaluation in the context θ (in symbols $[\![trm]\!]_\theta$);
- a satisfaction relation $\vDash$ such that, given a guard (or a trigger) β, $\theta \vDash \beta$ if the guard (or the trigger) is satisfied in context θ;
- an action-application function $\theta \xrightarrow{\alpha} \theta'$ mapping a context θ and a ground action α to the evaluation context θ' resulting from the execution of α on θ.

In the following, a formal definition for such elements is provided after briefly introducing some new necessary notation. Given a sequence $s = (s_1, s_2, \ldots, s_n)$, $head(s)$ denotes its first element s_1 and $tail(s) = (s_2, \ldots, s_n)$ denotes the sequence obtained from s by removing its first element. Consider a function $f : X \to Y$ and two elements x, y such that $x \in X$ and $y \in Y$. $f[x := y] = f'$ denotes a function such that $f'(x) = y$ and $f'(z) = f(z)$ for all $z \in X, z \neq x$.

Definition 4 (DSTM ground term denotation function)**.** Given an evaluation context $\theta = \langle \rho, \chi, \eta \rangle$, a term denotation function is a function $[\![\cdot]\!]_\theta : trm_P \to \mathcal{D}(\mathtt{T})$ recursively defined as follows:

- $[\![d]\!]_\theta = d$ if $d \in \mathcal{D}(\mathtt{Int})$;
- $[\![\mathtt{Chn}{::}c]\!]_\theta = c$ if $c \in \mathcal{D}(\mathtt{Chn})$;
- $[\![T_i{::}d]\!]_\theta = d$ if $d \in \mathcal{D}(T_i)$;
- $[\![x]\!]_\theta = \rho(x)$ if $x \in X$;
- $[\![len(c)]\!]_\theta = |\chi(c)|$ if $\chi(c) \neq \{\bot\}$, 0 otherwise;
- $[\![\boxdot_1 trm]\!]_\theta = \boxdot_1 [\![trm]\!]_\theta$;
- $[\![trm_1 \boxdot_2 trm_2]\!]_\theta = [\![trm_1]\!]_\theta \boxdot_2 [\![trm_2]\!]_\theta$.

with trm, trm_1, trm_2 being ground terms.

Definition 5 (DSTM ground trigger satisfaction relation)**.** Given an evaluation context $\theta = \langle \rho, \chi, \eta \rangle$, a ground trigger satisfiability relation is defined recursively as follows:

- $\theta \vDash \tau$;
- $\theta \vDash c?$ if $\chi(c) \neq \{\bot\}$;
- $\theta \vDash c?T_i$ if $head(\chi(c)) \in \mathcal{D}(T_i)$ and either $type(\hat{c}) = T_i$ if $type(\hat{c})$ is a simple type or $T_i \in type(\hat{c})$ if $type(\hat{c})$ is a multi-type.
- $\theta \vDash \xi_1 \vee \xi_2$ if $\theta \vDash \xi_1$ or $\theta \vDash \xi_2$;
- $\theta \vDash \xi_1 \wedge \xi_2$ if $\theta \vDash \xi_1$ and $\theta \vDash \xi_2$;
- $\theta \vDash \neg\xi$ if $\theta \nvDash \xi$;

with $c \in C$ being a channel and ξ, ξ_1, ξ_2 ground triggers.

Definition 6 (DSTM ground guard satisfaction relation)**.** Given an evaluation context $\theta = \langle \rho, \chi, \eta \rangle$, a ground guard satisfiability relation is defined recursively as follows:

- $\theta \vDash True$;
- $\theta \vDash c[?\top]$ if $|\chi(c)| = bd(c)$;
- $\theta \vDash c[?\bot]$ if $\chi(c) = \{\bot\}$;
- $\theta \vDash c[?\langle trm_1, \ldots, trm_k \rangle]$ if $|\chi(c)| \geq 1$ and $head(c) = \langle t_1, \ldots, t_k \rangle$ and for each $i \in \{1, \ldots, k\}$ either $t_i = [\![trm_i]\!]_\theta$ or $trm_i = \_$;
- $\theta \vDash trm_1 = trm_2$ if $Type(trm_1) = Type(trm_2)$ and $[\![trm_1]\!]_\theta = [\![trm_2]\!]_\theta$;
- $\theta \vDash trm_1 \odot trm_2$ if $Type(trm_1) = Type(trm_2) = \texttt{Int}$ and $[\![trm_1]\!]_\theta \odot [\![trm_2]\!]_\theta$;
- $\theta \vDash \phi_1 \vee \phi_2$ if $\theta \vDash \phi_1$ or $\theta \vDash \phi_2$;
- $\theta \vDash \phi_1 \wedge \phi_2$ if $\theta \vDash \phi_1$ and $\theta \vDash \phi_2$;
- $\theta \vDash \neg\phi$ if $\theta \nvDash \phi$;

with $c \in C$ being a channel, $\odot \in \{\leq, \geq\}$, $trm_1, \ldots, trm_k$ ground terms and ϕ, ϕ_1, ϕ_2 ground guards.

With all these elements in place, it is finally possible to define the action-application function $\xrightarrow{\alpha}$ as follows.

Definition 7 (Action application function)**.** Let $\theta = \langle \rho, \chi, \eta \rangle$ be an evaluation context and let $\alpha \in \mathcal{A}_P$ be a well-formed ground action. The action application function is defined recursively on the length of the sequence of atomic actions α as follows:

- if $\alpha = \varepsilon$, then $\theta \xrightarrow{\alpha} \theta$;
- if $\alpha = a; \alpha'$ for some atomic action a, then $\theta \xrightarrow{\alpha} \theta''$ if $\theta \xrightarrow{a} \theta'$ and $\theta' \xrightarrow{\alpha'} \theta''$;
- if $\alpha = a$ for some atomic action a, then:
  - if a is an assignment action of the form $x := trm$, then $\theta \xrightarrow{a} \langle \rho', \chi, \eta \rangle$, with $\rho' = \rho[x := [\![trm]\!]_\theta]$;
  - if a is a read action of the form $c[?]\langle v_1, \ldots, v_k \rangle$, then $\theta \xrightarrow{a} \langle \rho', \chi, \eta \rangle$, where, for each $x \in X$, $\rho'(x) = (head(c))_i$ if $x = v_i$ for some $i \in \{1, \ldots, k\}$ and $|\chi(c)| > 0$, $\rho(x)$ otherwise;
  - if a is a read action of the form $c?\langle v_1, \ldots, v_k \rangle$, then $\theta \xrightarrow{a} \langle \rho', \chi', \eta \rangle$, with ρ' defined as in the previous case and, for each $c' \in C$, $\chi'(c') = tail(c)$ if $c' = c$, $c \in C_I$ and $|\chi(c)| > 0$, $\chi(c')$ otherwise;
  - if a is a send action of the form $c!\langle trm_1, \ldots, trm_k \rangle$, then $\theta \xrightarrow{a} \langle \rho, \chi', \eta' \rangle$, where
    - if $c \in C_I$ is an internal channel then $\eta' = \eta$ (η in defined only on external channels) and $\chi' = \chi$ if $|\chi(c)| = bd(c)$ (i.e. the channel is full), $\chi[c := \chi(c) \cdot \langle [\![trm_1]\!]_\theta, \ldots, [\![trm_k]\!]_\theta \rangle]$ otherwise.
    - if $c \in C_E$ is an external channel then $\chi' = \chi$ (because signals sent over external channels cannot trigger transitions in the same step but only in the next one) and $\eta' = \eta$ if $\eta(c) \neq \{\bot\}$ (i.e. the channel is full), $\eta[c := \langle [\![trm_1]\!]_\theta, \ldots, [\![trm_k]\!]_\theta \rangle]$ otherwise.

1.3.2 Machine instantiation

Let P be a set of parameters, $\ell \in \Upsilon_P$ a ground parameter-substitution function whose domain is P and whose co-domain is a set of ground terms. Let trm be a term such that the set $Params(trm)$ of parameters occurring in it is included in P. $trm[\ell]$ is the ground term obtained by substituting each occurrence of a parameter p in trm with its denotation $[\![\ell(p)]\!]_\theta$. In a similar way, given an action $\alpha \in \mathcal{A}_P$ (resp. a trigger $\xi \in \Xi_P$, a guard $\phi \in \Phi_P$), $\alpha[\ell]$ is defined as the action obtained from α (resp. ξ, ϕ) by substituting each term trm in it with $trm[\ell]$.

Yet parameter-substitution functions need not to be ground and could substitute a parameter with a non-ground term. As an example, consider the *Counting* DSTM specification detailed in Figure 1.1 and in Table 1.1: the parameter-substitution functions on transitions T7 and T8 are not ground, as they assign to *Incrementer*'s parameter `P_limit` a non-ground term formed by *Counter*'s parameter `P_to`. When an *Incrementer* machine is instantiated by a *Counter*, however, parameter `P_to` must have been actualized to some ground term, so the "overall" parameter-substitution is indeed ground. To formalize this behaviour, it is necessary to extend the application of a ground substitution ℓ to a non-necessarily ground substitution ℓ'. Let ℓ' be a parameter-substitution function and $Params(\ell') = \{p \mid p \in Params(\ell'(p')), p' \in P\}$ be the set of parameters occurring in

the image of ℓ'. Assume $Params(\ell') \subseteq P$. Then $\ell'[\ell]$ is a ground parameter-substitution function such that $\ell'[\ell](p') = \ell'(p')[\ell]$. Continuing with the previous example, let $\ell = \{(\texttt{P\_to}, 100)\}$ be the parameter-substitution function associated with transition T2 (i.e. $Inst_1(\texttt{T2}) = \ell$) and let $\ell' = \{(\texttt{P\_limit}, \texttt{P\_to})\}$ be the parameter-substitution function associated with transition T7. $Params(\ell') = \{\texttt{P\_to}\}$, while ℓ is ground and $Params(\ell) = \emptyset$. $\ell'[\ell]$ is a ground substitution function as $\ell'[\ell](\texttt{P\_limit}) = \ell'(\texttt{P\_limit})[\ell] = \texttt{P\_to}[\ell] = [\![\ell(\texttt{P\_to})]\!]_\theta = 100$.

At last, it is now possible to define the ground instantiation $M[\ell]$ of a parametric machine M with regard to the parameter-substitution function ℓ as the machine obtained from M by substituting each action α, guard ϕ, trigger ξ and parameter-substitution function ℓ' with the corresponding ground objects $\alpha[\ell]$, $\phi[\ell]$, $\xi[\ell]$ and $\ell'[\ell]$.

1.3.3 Semantics by means of a Labelled Transition System

In this subsection formal semantics for DSTM is provided by defining a Labelled Transition System (LTS). An LTS, often used to describe the potential behaviour of systems, is a 4-ple $L = \langle S, \Sigma, \Delta, S_0 \rangle$, where:

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

The main intuition behind this formalization is that each state $s \in S$ of the LTS model represents a complete configuration (*state*) of the DSTM in a given instant, including the current control locations and the current evaluation context, while a *step* in the DSTM will correspond to a suitably-defined series of LTS transitions, each capturing one DSTM transition or more.

Let us begin with the problem of representing the complete configuration of a DSTM in a state of the LTS. To represent the current control locations it is necessary to store information about the current state each currently-active process (ground machine) is in, and the information about the activating processes (calling ground machines). Since a machine may instantiate multiple machines, the information about the current control locations can be represented with a tree. Each vertex of such tree, called the *control tree*, is labelled with either a machine, a box or a node. Accordingly to the intuition that pseudo-nodes represent only transient non-stable control points, control tree vertices cannot be labelled by pseudo-nodes. The root of a *control tree*, labelled by a machine, represents the main (initial) process, having the highest level in the hierarchy. Leaves represent control states in which each currently-active process is in and are labelled by nodes. Internal vertices represent the call hierarchy and cannot be labelled by nodes. Whenever a vertex is labelled by a machine M, it either is the root or is the child of a node

labelled by a box instantiating M. If a node is labelled by either a box or a node, then its parent is labelled by the machine to which the box or the node belong.

A labelled tree $\mathcal{T}$ is a pair $\langle Vx, \lambda \rangle$ where $Vx \subseteq \mathbb{N}^*$ is a prefix-closed set of vertices (i.e. if $v \in Vx$ and $v = v' \cdot v$, then $v' \in Vx$), with the empty sequence denoted by ε, and $\lambda : Vx \longrightarrow \Gamma$ is a labelling function mapping each vertex to some label set Γ. Each vertex (string) in a similarly-defined tree encodes a path from the root of the control tree to that vertex itself: ε encodes the empty path, so the root, i identifies the path to the i-th child of the root, $i \cdot j$ encodes the path to the j-th child of the i-th child of the root, and so on. For the sake of brevity, given a DSTM $D = \langle M_1, \ldots, M_n, X, C, P \rangle$, let $N(D)$ denote the containing of all nodes of each machine, namely $\bigcup_{i=1}^{n} N_i$. Similarly, for boxes and machines, let $Bx(D) = \bigcup_{i=1}^{n} Bx_i$ and $M(D) = \bigcup_{i=1}^{n} \{M_i[\ell] \mid \ell \in \Upsilon_{P_i}\}$.

Definition 8 (Control tree over a DSTM)**.** A *control tree* over the DSTM $D = \langle M_1, \ldots, M_n, X, C, P \rangle$ is a labelled tree $\langle Vx_{ct}, \lambda \rangle$ where Vx_{ct} is a set of vertices such that each vertex is parent of at most one leaf and $\lambda : Vx_{ct} \longrightarrow N(D) \cup Bx(D) \cup M(D)$ is a labelling function such that

- $\lambda(\varepsilon) = M_1$;
- $n \in Leaves(Vx_{ct})$ iff $\lambda(n) \in N(D)$;
- if $n = n' \cdot i$ with $i \in \mathbb{N}$ (i.e. n is the i-th child of n'), then
  - (i) if $n \in Leaves(Vx_{ct})$, then $\lambda(n) \in N_j$ and $\lambda(n') \in M_j$ for some $j \in \{1, \ldots, n\}$;
  - (ii) if $n \notin Leaves(Vx_{ct})$ and $\lambda(n) = bx \in Bx_j$ for some $j \in \{1, \ldots, n\}$ then $\lambda(n') \in M_j$;
  - (iii) if $n \notin Leaves(Vx_{ct})$ and $\lambda(n) = M_j$ for some $j \in \{1, \ldots, n\}$ then $\lambda(n') = bx \in Bx_k$ for some $k \in \{1, \ldots, n\}$ and $j \in Y_k(bx)$;

In Definition 8, constraint (i) and (ii) require for each node-labelled or box-labelled vertex to be child of a vertex labelled with the machine to which the node belongs, constraint (iii) requires for each machine-labelled node to be the child of a vertex labelled with a box instantiating the machine.

Example 6 (Control trees for the *Sample* DSTM)**.** Consider the *Sample* DSTM depicted in Figure 1.1 and detailed in Table 1.1 and throughout examples 1, 2. Some of the *Sample* DSTM's possible control trees are represented in Figure 1.3. In the figure, each machine-labelled vertex is depicted as a diamond ($\diamond$), each box-labelled vertex as a square, and each node-labelled vertex as a circle (crossed-out if it is labelled by a final node). On the right of each node n, the value of $\lambda(n)$ is shown.

Tree (a) encodes the control state in which only the *Main* machine is running and is in the idle1 state.

Tree (b) encodes the control state in which the *Main* machine has entered the box counterBox, thus instantiating an instance of the *Counter* machine which is in its state idle2.

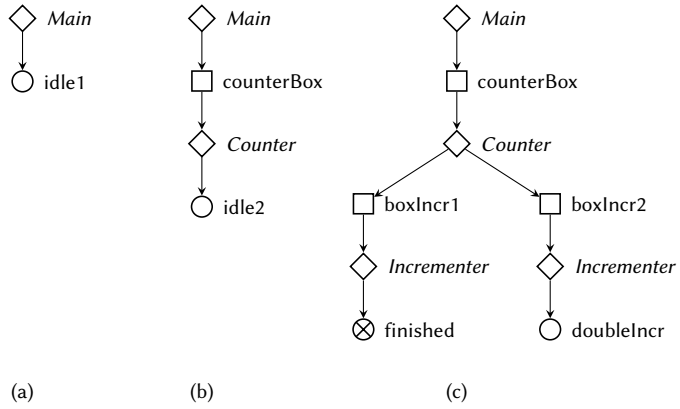


Figure 1.3: Control trees of the *Sample* DSTM

Tree (c) encodes the control state in which the *Counter* machine, instantiated by *Main* by entering the box counterBox, in turn instantiated two *Incrementer* machines by entering the boxes boxIncr1 and boxIncr2, with the *Incrementer* machines being in the finished and doubleIncr state respectively.

Definition 9 (DSTM state)**.** The state of a DSTM D is a tuple $\langle CT, Fr, \theta \rangle$ where

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

For the sake of clarity, the ground guard/trigger satisfaction relation can be extended to states as follows: given a state $s = \langle CT, Fr, \theta \rangle$ and a ground guard ϕ (resp. ground trigger ξ), $s \vDash \phi$ (resp. $s \vDash \xi$) if $\theta \vDash \phi$ (resp. $\theta \vDash \xi$). For each implicit transition $t \in T_i$ of a ground machine M_i, with $Src_i(t) = en \in En_i$ and $Trg_i(t) = n \in N_i$, we define $explicit_{M_i}(en) = n$.

The set of initial states S_0 of the LTS contains states $s_0 = \langle CT_0, Fr_0, \theta_0 \rangle$, where

- $CT_0 = \langle Vx_0, \lambda_0 \rangle$, with $Vx_0 = \{\varepsilon, 1\}$, $\lambda_0(\varepsilon) = M_1$, $\lambda_0(\varepsilon) = explicit_{M_1}(df_1)$;
- $Fr_0 = Vx_0$;
- the initial evaluation context $\theta_0 = \langle \rho_0, \chi_0, \eta_0 \rangle$ is such that:
  - the variable evaluation function ρ_0 assigns to each variable the default value for its type, i.e. $\rho_0(x) = default(type(x))$ for each $x \in X$;
  - χ_0 assigns $\perp$ to each internal channel and a non-deterministically chosen value (including $\perp$) to each external channel;
  - η_0 assigns $\perp$ to each external channel.

Notice that control tree (a) in Figure 1.3 is a valid initial control tree for the *Sample* DSTM.

With the states S and the initial states S_0 defined, what remains is to define transitions. To do so, it is necessary to generalize the notion of DSTM transition by introducing *compound transitions*. Given a machine M_i, a compound transition is a pair of sequences of transitions $ct = \langle\langle t_1, \ldots, t_j\rangle, \langle t'_1, \ldots, t'_k\rangle\rangle$ with $\{t_1, \ldots, t_j, t'_1, \ldots, t'_k\} \in T_i$. $t_1, \ldots, t_j$ are called *incoming transitions* and $t'_1, \ldots, t'_k$ are called *outgoing transitions*. DSTM transitions may have source or target in pseudo-nodes which, as said, correspond to transient, unstable control points. Therefore, a transition involving pseudo-nodes may be seen as part of a *super-transition* connecting proper control points. For example, a fork (resp. a join) can be seen as a super-transition connecting one source with multiple targets (resp. multiple sources with one target). *Compound transitions* are able to capture this intuition and allow us to consider only transitions having source(s) and target(s) in proper control points.

A compound transition ct is decorated by a trigger $\xi(ct)$, a guard $\phi(ct)$ and by two sets of actions $\alpha^{pre}(ct)$ and $\alpha^{post}(ct)$, collecting respectively the actions of incoming and outgoing transitions. There exist three types of compound transitions:

simple having the form $ct = \langle\langle t\rangle, \langle t\rangle\rangle$, with t being a non-implicit transition such that neither its source or its target are fork or join nodes.

fork having the form $ct = \langle\langle t\rangle, \langle t'_1, \ldots, t'_k\rangle\rangle$ and such that there exists a fork pseudo-node $fk \in Fk_i$ and $Trg_i(t) = fk$ and $\{t'_1, \ldots, t'_k\}$ is the set of all the transitions having as source either fk or $(fk, \downarrow)$. In this case, $\xi(ct) = \xi(t)$ and $\phi(ct) = \phi(t)$ (all transitions exiting from a fork must have the trivial trigger τ, as per table 1.2), $\alpha^{pre}(ct) = \alpha(t)$ and $\alpha^{post}(ct) = \prod_{\ell=1}^{k} \alpha(t'_\ell)$.

join having the form $ct = \langle\langle t_1, \ldots, t_j\rangle, \langle t'\rangle\rangle$ and such that there exists a join pseudo-node $jn \in \mathcal{J}n_i$ and $Trg_i(t') = jn$ and $\{t_1, \ldots, t_j\}$ is the set of all the transitions having as target either jn or $(jn, \otimes)$. In this case, $\xi(ct) = \bigwedge_{i=1}^{j} \xi(t_i)$ and $\phi(ct) = \bigwedge_{i=1}^{j} \phi(t_i)$, $\alpha^{pre}(ct) = \varnothing$ (entering join actions are not allowed to contain non-empty actions) and $\alpha^{post}(ct) = \alpha(t')$. Moreover, if a transition $t \in \{t_1, \ldots, t_j\}$ has as target $(jn, \otimes)$, then the whole compound transition is a preemptive join and $\otimes(ct) = Src_i(t)$.

Semantics of transitions require to formalize: (i) when a compound transition is enabled in a given state $s \in S$; (ii) what changes are induced on the starting state by the execution of a compound transition.

Let us start by defining the notion of *enabled-ness* of a transition w.r.t. a state $s \in S$. Such enabled-ness depends only on the form of the source vertex of the transition. Formally, given a state $s = \langle CT = \langle Vx, \lambda\rangle, Fr, \theta\rangle$, a transition t and a vertex $n \in Vx$, a predicate $Enabled(t, n)$, meaning that t is enabled in vertex n, is introduced and defined as follows: $s \vDash Enabled(t, n)$ iff the subtree of CT rooted in n is contained within the frontier Fr and one of the following conditions holds, depending on the type of t:

basic transitions (namely transitions whose source is a node, e.g. internal, entering join, entering fork, call transitions, etc.) $Src_i(t) \in N_i$, $n \in Leaves(Vx)$, $\lambda(n) = Src_i(t)$, $s \vDash \xi(t)$ and $s \vDash \phi(t)$;

return by interrupt $Src_i(t) \in Bx_i$, $\xi(t) \neq \tau$, $\lambda(n) = Src_i(t)$, $s \vDash \xi(t)$ and $s \vDash \phi(t)$;

return by default $Src_i(t) \in Bx_i$, $\xi(t) = \tau$, $\lambda(n) = Src_i(t)$, $s \vDash \phi(t)$ and $\lambda(n \cdot j \cdot 1) \in Ex(D)$ for all $n \cdot j \cdot 1 \in Vx$ (i.e. for each j-th machine instantiated by the box $\lambda(n)$, its child is labelled with an exit state);

return by exiting $Src_i(t) = (bx, ex)$, with $bx \in Bx_i$ and $ex \in Ex_{Y_i(bx)}$, $\lambda(n) = bx$ and $n \cdot 1 \cdot 1 \in Vx$, with $\lambda(n \cdot 1 \cdot 1) = ex$ (notice that return by exiting transitions are only allowed from boxes instantiating exactly one machine and must have trivial trigger and guard, so it is not necessary to require that they are satisfied).

The notion of enabled-ness is extended to compound transitions as follows. Given a state $s = \langle CT = \langle Vx, \lambda \rangle, Fr, \theta \rangle$, a set of k sibling vertices $N = \{n \cdot i_1, \ldots, n \cdot i_j\} \subset Vx$ and a compound transition $ct = \langle \langle t_1, \ldots, t_j \rangle, \langle t'_1, \ldots, t'_k \rangle \rangle$, $s \vDash Enabled(ct, N)$ iff:

(i) $\lambda(n) = M_i$;

(ii) for all $\overline{n} \in N$ and compound transition ct', if $N' \subset Vx$, $n' \in N'$ and $n' \prec \overline{n}$ (with $n' \neq \overline{n}$ and $\prec$ being the canonical *prefix* relation defined on strings), then $s \nvDash Enabled(ct', N')$;

(iii) if ct is not a preemptive join, then for all $\ell \in \{1, \ldots, j\}$ $s \vDash Enabled(t_\ell, n \cdot i_\ell)$;

(iv) if ct is a preemptive join, then for all $\ell \in \{1, \ldots, j\}$

- if $t_\ell \in \otimes(ct)$, then $s \vDash Enabled(t_\ell, n \cdot i_\ell)$;
- if $Src_i(t_\ell) = (bx, ex)$, then $\lambda(n \cdot i_\ell) = bx$, otherwise $\lambda(n \cdot i_\ell) = Src_i(t_\ell)$.

Condition (i) requires that the parent vertex n of the vertices in N is labelled with the same machine to which ct belongs. Condition (ii) requires that no ancestor n' of the nodes in N is involved in an enabled (and potentially interrupting) compound transition. This condition guarantees that an interrupting transition enabled in some internal node of the tree has higher priority over all transitions enabled in nodes of its subtree. Condition (iii) requires that, when ct is not a preemptive join, all of the incoming transitions are enabled. Condition (iv) deals with preemptive joins and only requires that the incoming interrupting transition ($\otimes(ct)$) is enabled and that all other incoming transitions have properly-labelled sources.

The execution of a compound transition has effects on the structure of the control tree and on the evaluation context of the state from which it is taken. Given a state s and a compound transition $ct = \langle \langle t_1, \ldots, t_j \rangle, \langle t'_1, \ldots, t'_k \rangle \rangle$ belonging to some machine M_i, in order to account for the new subtrees to be added to the current control tree, a sequence of trees $Trees(ct, s) = \{CT_1, \ldots, CT_k\}$ is defined as follows. For each t'_m, $\ell \in \{1, \ldots, k\}$

- if $Trg_i(t'_m) = n \in N_i$, then the control tree $CT_m = \langle Vx_m, \lambda_m \rangle$ with $Vx_m = \{\varepsilon\}$ and $\lambda_m(\varepsilon) = n$;

- if $Trg_i(t'_m) = (bx, en)$, with $bx \in Bx_i$ and $en \in En_{Y_i(bx)}$, then the control tree $CT_m = \langle Vx_m, \lambda_m \rangle$ with $Vx_m = \{\varepsilon, 1, 1 \cdot 1\}$ and $\lambda_m(\varepsilon) = bx$, $\lambda_m(1) = M_h[\ell]$, with $Y_i(bx) = h$ and $\ell = Inst_i(t'_m)$, and $\lambda_m(1 \cdot 1) = explicit_{M_h}(en)$;
- if $Trg_i(t'_m) = bx$, with $bx \in Bx_i$, then the control tree $CT_m = \langle Vx_m, \lambda_m \rangle$ with $Vx_m = \{\varepsilon\} \cup \bigcup_{z=1}^{|Y_i(bx)|} (\{z\} \cup \{z \cdot 1\})$ and, for all $1 \leq z \leq |Y_i(bx)|$, $\lambda_m(\varepsilon) = bx$, $\lambda_m(z) = M_{h_z}[\ell_z]$, $\lambda_m(z \cdot 1) = explicit_{M_{h_z}}(df_1)$ with $h_z = (Y_i(bx))_z$ and $\ell_z = (Inst_i(t'_m))_z$.

In order to describe the evolution of a control tree induced by the firing of a compound transition it is necessary to introduce suitable tree transformation operations. Given two labelled trees $\mathcal{T} = \langle Vx, \lambda \rangle$, $\mathcal{T}' = \langle Vx', \lambda' \rangle$ and a node $n \in Vx$, the operation $\mathcal{T} \circ_{n \cdot i} \mathcal{T}'$ produces the tree $\mathcal{T}'' = \langle Vx'', \lambda'' \rangle$ obtained from T by replacing the (possibly empty) subtree rooted in $n \cdot i$ with $\mathcal{T}'$. Formally, $\mathcal{T}'' = \mathcal{T} \circ_{n \cdot i} \mathcal{T}'$ iff for all vertices $n'' \in Vx''$, one of the following is satisfied:

- $n'' \in Vx$, $n \cdot i \not\prec n''$ and $\lambda''(n'') = \lambda(n'')$ (n'' was not part of the subtree of $\mathcal{T}$ rooted in $n \cdot i$ and is not affected by any modification);
- $n'' = n \cdot i \cdot n'$ for some $n' \in Vx'$ and $\lambda''(n'') = \lambda'(n')$ (n'' was part of $\mathcal{T}'$, is now a descendant of $n \cdot i$ and maintains its original labelling).

Given a labelled tree $\mathcal{T} = \langle Vx, \lambda \rangle$ and a set of nodes $N \subseteq Vx$, the operator $Remove(\mathcal{T}, N)$ produces the tree $\mathcal{T}' = \langle Vx', \lambda' \rangle$ obtained from $\mathcal{T}$ by removing all the nodes belonging to a subtree rooted in one of the nodes in N and by leaving the labelling unchanged for the remaining nodes. Formally, $n' \in Vx'$ iff $n' \in Vx$ and $n \not\preceq n'$ for all $n \in N$ (i.e. a vertex is in $\mathcal{T}'$ iff it was in $\mathcal{T}$ and is not a descendant of any node in N) and $\lambda' = \lambda \restriction_{Vx'}$, where $\lambda \restriction_{Vx'}$ denotes the restriction of λ to Vx'.

The *Update* operator, taking as arguments a tree $\mathcal{T}$, a node n in $\mathcal{T}$ and a possibly empty sequence of trees $\mathcal{T}_1, \ldots, \mathcal{T}_k$, is defined inductively on the length of the sequence as follows:

- $Update(\mathcal{T}, n, \varepsilon) = \mathcal{T}$;
- $Update(\mathcal{T}, n, \langle \mathcal{T}_1, \ldots, \mathcal{T}_n \rangle) = Update((\mathcal{T} \circ_{n \cdot i} \mathcal{T}_1), n, \langle \mathcal{T}_2, \ldots, \mathcal{T}_n \rangle)$, with i being the smallest number such that $n \cdot i \notin \mathcal{T}$.

For a transition $t \in T_i$, $\alpha_{impl}(t)$ is the set of all *implicit actions* associated with implicit transitions triggered by the execution of t. If t is a call by default transition, $\alpha_{impl}(t)$ is the set of all actions associated with the implicit transitions having as source the default entering node for each machine instantiated by the box t is entering. In the same way, if t is a call by entering having as target (bx, en), $\alpha_{impl}(t)$ is the action decorating the implicit transition from *en*. Formally,

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

Finally, it is possible to provide the semantics of a compound transitions $ct = \langle\langle t_1, \ldots, t_j\rangle, \langle t'_1, \ldots, t'_k\rangle\rangle$ w.r.t a certain state $s = \langle \mathcal{T}, Fr, \theta\rangle$. Assuming that ct is enabled in s, i.e. there is a set N of j siblings in Fr such that $s \vDash Enabled(ct, N)$ and these vertices in N are the sources of the incoming transitions, the successor state s' resulting from the execution of ct from s is obtained as follows:

- the control state is obtained by removing from $\mathcal{T}$ the subtrees rooted in nodes in N ($Remove(\mathcal{T}, N)$) and then by adding the control trees in $Trees(ct, s)$ as additional children of the parent n of the siblings in N ($Update(\mathcal{T}, n, Trees(ct, s))$).
- as regards data-flow modifications, the new evaluation context θ' is obtained by applying incoming actions, then outgoing actions and then the possible implicit actions triggered by the outgoing transitions. Notice that incoming transitions, as well as outgoing and implicit transitions, may be executed in any possible order since they are performed by concurrent processes. So, every permutation of incoming actions followed by any permutation of outgoing actions followed by any permutation of implicit actions is a possible outcome for the execution of ct.

Formally, the transition relation is defined as follows.

Definition 10 (DSTM semantics - transition relation)**.** Given a state $s = \langle \mathcal{T}, Fr, \theta\rangle$ and a compound transition belonging to some machine M_i,

$$\langle \mathcal{T}, Fr, \theta\rangle \xrightarrow{ct} \langle \mathcal{T}', Fr', \theta'\rangle$$

iff there exists a vertex n and a set $N \subseteq Fr$ of children of n such that

(i) $s \vDash Enabled(ct, N)$;

(ii) $\mathcal{T}' = Update\,(Remove(\mathcal{T}, N), n, Trees(ct, s))$;

(iii) $Fr' = Fr \setminus \{n' \in Fr \mid n \preceq n' \text{ for some } n \in N\}$;

(iv) $\theta \xrightarrow{\alpha_1;\alpha_2} \theta'$, where $\alpha_1 = \alpha^{pre}(ct)$ and

- if ct is a compound fork transition of the form $\langle\langle t\rangle, \langle t'_1, \ldots, t'_k\rangle\rangle$ then α_2 is a permutation of the actions in

$$\bigcup_{j=1}^{k} \left(\{\alpha(t_j)\} \cup \{\alpha[\ell_j] \mid \alpha_{impl}(t'_j) = \{\alpha\} \text{ and } \ell_j = Inst_i(t'_j)\} \right)$$

- if ct is a join compound transition of the form $\langle\langle t_1, \ldots, t_j\rangle, \langle t\rangle\rangle$ or a simple transition of the form $\langle\langle t\rangle, \langle t\rangle\rangle$, then $\alpha_2 = \overline{\alpha}; \alpha$ where $\overline{\alpha}$ is any permutation of the transitions in $\alpha^{post}(ct)$ and α is defined as follows:

- if t is a call by default transition entering a box b, then α is a permutation of the set

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

- if t is a call by entering transition entering a box b and specifying entering state *en*, then $\alpha = \alpha\left(\bar{t}_j\right)\left[\ell_j\right]$, with $\alpha\left(\bar{t}_j\right) = \alpha_{impl}(t)$, $Src_z\left(\bar{t}_j\right) = en$ and $\ell_j = (Inst_i(t))$;
- $\alpha = \epsilon$, otherwise.

When no more "internal" transitions can be performed (depending on the current control tree, frontier and evaluation context), the LTS performs a transition

$$\langle \mathcal{T}, Fr, \theta \rangle \xrightarrow{next} \langle \mathcal{T}, Fr', \theta' \rangle$$

corresponding to the completion of the current step and the initialization of the next step, where $\mathcal{T} = \langle Vx, \lambda \rangle$, $Fr' = Vx$, $\theta = \langle \rho, \chi, \eta \rangle$ e $\theta' = \langle \rho, \chi', \varnothing \rangle$ such that

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

During this next-step initialization transition the context tree remains unchanged, the frontier is changed to all vertices in the context tree, variable and internal channel evaluation do not change and the content of the external channel c is either set to match any non-empty message specified by the η function or non-deterministically initialized if $\eta(c) = \perp$.

Example 7 (Steps in a *Dynamic* DSTM computation)**.** Consider the *Dynamic* DSTM detailed in Figure 1.2 and in Table 1.3. Figure 1.4 shows steps in one of its possible computations, as elaborated below. In its initial state s_0, the DSTM has a control state encoded by tree (a). Suppose that the external environment generates a message on the external `req` channel, thus enabling transition T2. By the definition of enabled-ness for compound transitions, the compound asynchronous fork $ct_1 = \langle \langle \mathsf{T2} \rangle, \langle \mathsf{T3}, \mathsf{T4} \rangle \rangle$ is therefore enabled in the waiting-labelled node. No other compound transitions are enabled, so the first step consists only in ct_1 and in the re-initialization transition *next*. When ct_1 fires, the node 1 (labelled by waiting) is removed from control tree (a) by calling *Remove*((a), {1}) and subsequently the new trees in $Trees(ct_1, s_0)$ are added to the resulting control tree by calling $Update(Remove((a), \{1\}), \varepsilon, Trees(ct_1, s_0))$ and obtaining tree (b). Suppose that another message is available on the external channel `req`. Compound transition ct_1 is again enabled in node 1. This time also T14 from the *Incrementer* machine is enabled, and so is the simple compound transition $ct_2 = \langle \langle \mathsf{T14} \rangle, \langle \mathsf{T14} \rangle \rangle$. The second step consists of two compound transitions ct_1 and ct_2, which may be executed in any order, followed as always by the step initialization transition. Execution of step 2 results in the control tree (c),

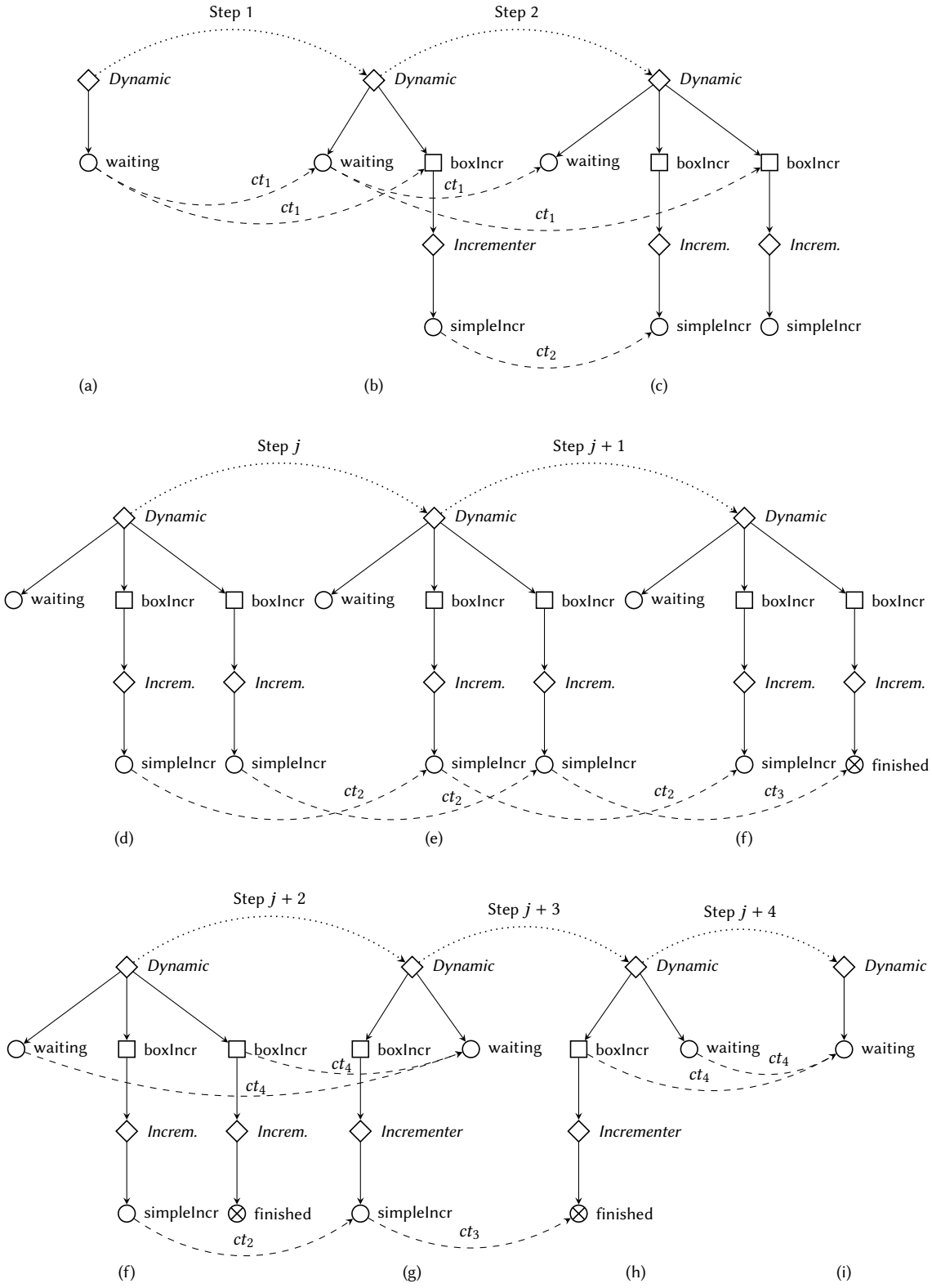


Figure 1.4: Steps in a *Dynamic* DSTM computation

where two instances of the *Incrementer* machine are executing concurrently along with the *Dynamic* machine, which is waiting for new requests in its waiting state. From now on, no more requests arrive on the external channel and the two concurrent *Incrementer* machines continue incrementing the global variable `x`. The system evolves by performing steps like step j, consisting in two compound transitions ct_2 firing from the simpleIncr states in the two concurrent *Incrementers*. Eventually, `x` will be incremented enough to be greater or equal than the `P_limit` parameter in one of the *Incrementers*. In the scenario depicted in Figure 1.3, this happens for the second machine after the execution of step j. Therefore, in step $j+1$, transition T14 is no longer enabled but transition T16 is enabled in node $3 \cdot 1 \cdot 1$. Step $j+1$ consists in a ct_2 transition from $2 \cdot 1 \cdot 1$ and in a $ct_3 = \langle\langle \mathsf{T16} \rangle, \langle \mathsf{T16} \rangle\rangle$ transition from $3 \cdot 1 \cdot 1$ and results in control tree (f). In (f), the compound join transition $ct_4 = \langle\langle \mathsf{T5}, \mathsf{T6} \rangle, \langle \mathsf{T7} \rangle\rangle$ is enabled in nodes $\{1, 3\}$, as well as the ct_2 transition is enabled in $\{2 \cdot 1 \cdot 1\}$. Hence, step $j+2$ consists in the compound transitions ct_4,ct_2, again executed in any possible order, and results in control tree (g). In (g) `x` is finally greater or equal than the `P_limit` parameter passed to the first *Incrementer* and ct_3 is the only enabled compound transition for step $j+3$. In step $j+4$, only ct_4 is enabled and its execution results in tree (i).

–2–
Automatic test case generation from Dynamic State Machines

Contents: **2.1 The Spin model checker and Promela: a brief introduction.** 2.1.1 The Spin model checker – 2.1.2 The Promela specification language. **2.2 Deriving Promela models from DSTMs.** 2.2.1 An overview of the translation process. **2.3 Flattening the DSTM into ordinary state machines. 2.4 Promela encoding for the flat model.** 2.4.1 Translation of data-flow elements – 2.4.2 An overview of the Promela specification – 2.4.3 Mapping a flat DSTM to a Promela specification – 2.4.4 Enforcing the steps semantics – 2.4.5 Mapping a DSTM model to a Promela specification. **2.5 Test case generation.**

The DSTMs formalism, as described in detail in the previous chapter, is very expressive and well-suited to model complex systems in an insightful fashion. Because of these qualities, the formalism can be used in different phases of the development lifecycle ranging from design to verification and validation.

When dealing with verification and validation, it is possible to "translate" a DSTM model into a lower-level formalism – such as the specification language of a model checker – in order to enable property verification and/or test case generation. This approach has already been used with similar and less expressive formalisms such as statecharts [14, 15] and UML state machines [16].

This chapter describes an automatable process to translate a DSTM model into a Promela specification and a way to use such specification with the Spin model checker to derive test cases. After a brief introduction to the Spin model checker and its specification language Promela in Section 2.1, sections 2.2, 2.3, and 2.4 describe the translation process. Section 2.5 shows how to use the Promela specification to derive test cases covering specific transitions or nodes.

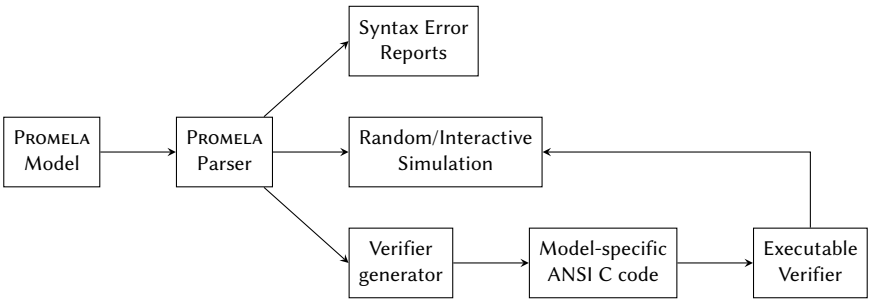


Figure 2.1: The structure of Spin [19]

2.1 The Spin model checker and Promela: a brief introduction

An exhaustive description of a complex tool like the Spin model checker and its rather rich verification language Promela is beyond the scope of this thesis work. This section briefly presents the Spin model checker and the Promela constructs used in the proposed translation in Section 2.2.

2.1.1 The Spin model checker

Spin [17] (Simple Promela INterpreter) is a verification tool for distributed and concurrent systems. It accepts design specifications written in the language Promela, and is able to execute them in simulation mode and produce an optimized on-the-fly verification program. Once the verification program, written in ANSI C, is compiled, it is used to perform the verification itself against correctness claims. In Spin, correctness claims are used to formalize undesired system behaviour. If any counterexample satisfying the correctness claims is detected, the verifier produces a trail of the run satisfying the claim (and exhibiting undesired behaviour). This trail can be fed back to Spin and inspected in detail in simulation mode to determine the cause of the violation [18]. The structure of Spin is shown in Figure 2.1.

Under the hood, as described by Holzmann in [19], Spin translates each concurrent process in the design specification to a finite automaton, then computes the asynchronous interleaving product of said automata to obtain an automaton representing the global state of the system. This interleaving product is often referred to as the *state space* of the system or as the *global reachability graph*. To perform verification, Spin proceeds as described by Vardi and Wolper in [20] and builds a Büchi automaton for the correctness claim – which typically is the negation of the property one wants to ensure –, then computes a *synchronous* product of the claim and the automaton representing the system obtaining – again – a Büchi automaton. If the language accepted by this Büchi automaton is empty, then there is no system run satisfying the correctness claim – i.e. a violation of the property –, otherwise a run belonging to said intersection is returned as a counterexample.

Type	Domain
`bit`	$\{0, 1\}$
`bool`	{true, false}
`byte`	$\{0, \ldots, 255\}$
`chan`	$\{1, \ldots, 255\}$
`mtype`	$\{1, \ldots, 255\}$
`pid`	$\{0, \ldots, 255\}$
`short`	$\{2^{15}, \ldots, 2^{15} - 1\}$
`int`	$\{2^{31}, \ldots, 2^{31} - 1\}$
`unsigned`	$\{0, \ldots, 2^n - 1\}$

Table 2.1: Basic data types in Promela

2.1.2 The Promela specification language

Promela (PROcess/PROtocol MEta LAnguage), the specification language for Spin, is designed to be an intuitive, program-like notation for specifying design choices unambiguously, without focusing on implementation details [19]. A Promela specification is a non-empty set of processes operating on data objects and communicating over message channels, with at least one initial process (namely a process required to be active in the initial system state). Each running process is an instantiation of a `proctype`, whose body contains data declarations and statements. `Proctypes` have global scope, while data objects can either have global scope, if declared outside of a `proctype`'s body, or process-local scope otherwise.

Since there is no general decision procedure for unbounded systems, Promela is designed in such a way that every model which can be specified is necessarily bounded, i.e. has a finite number of distinct states and behaviours [19, 18]. This guarantees that the *state space* can always be generated and explored in finite time (unfortunately, this does not mean that doing so is always feasible!).

Promela borrows many of its notation from the C programming language, including, for instance, the syntax for Boolean and arithmetic operators, for assignment (a single equals) and equality (a double equals), for variable and parameter declarations, variable initialization and comments, and the use of curly braces to indicate the beginning and end of program blocks [18].

In what follows, the main elements in Promela are briefly presented.

Datatypes and channels

In Promela there are only two levels of scope for data objects: global and process local. As previously said, data objects declared inside a `proctype`'s body are local to that process, while the others are global and visible in every process. Since there are only two levels of scope, there is no way to restrict the scope of a global variable to a given subset of processes and, similarly, it is not possible to restrict the scope of a local variable to a given

Listing 2.1 Variables and channels declarations

```
 1: bit foo, bar=1;
 2: mtype={white,red,green};       /* mtype declaration */
 3: mtype={blue,pink};             /* mtype declaration */
 4: mtype colour = red;
 5:
 6: typedef square {               /* user-defined type */
 7:     mtype colour[4];           /* array of 4 mtypes */
 8:     int sideSize;
 9: }
10: square sqArray[3];             /* 3 squares array   */
11: sqArray[0].colour[3] = red;
12:
13: chan a,b,c;                    /* channel declaration */
14: chan d = [2] of {mtype};       /* channel declaration and init. */
15: chan e = [32] of {mtype,bit}; /* messages with multiple fields */
```

block inside a process. All objects must be properly declared before being referenced. The basic data types in Promela are summarized in Table 2.1.

Variables of type `mtype` can hold symbolic values that must be introduced with one or more `mtype` declarations. Zero-indexed arrays can be declared as in the C language and the number of elements must be specified at declaration time with an integer constant. An array of `N` elements of type `t` is declared as `t name[N]`. User-defined datatypes, whose declaration via the `typedef` construct is required to have global scope, are also supported. Listing 2.1 shows some examples of variable and type declarations and assignments. Channels are declared (globally or locally) using the keyword `chan` and must be initialized before usage. A channel initialization specifies its length in square brackets as well as the type of the conveyed messages. Messages may also contain multiple fields, in which case a type for each field is specified in a bracket-enclosed comma-separated list as shown in Listing 2.1. By default, channels store messages in first-in-first-out order. In Listing 2.1 two `bit` variables `foo` and `bar` are declared, with the latter being initialized to `1`. In lines 2 and 3, two `mtype` declarations define five symbolic names for colours. Notice that those two declarations are in fact equivalent to a single declaration listing all five symbolic names. In line 4, the variable `colour` of type `mtype` is declared and initialized to `red`. Starting from line 6, the user-defined datatype `square` is declared, containing a `colour` field being an array of 4 `mtypes` and an `int` field named `sideSize`. In line 10 an array of 3 `squares` named `sqArray` is declared and line 11 sets a value for the last element in the `colour` field of the first `square` in the array.

Processes

In Promela, processes are declared with the `proctype` construct as shown in Listing 2.2. The keyword `active` can be prefixed to any proctype declaration, as in process `B` in the example, to define a set of processes that are required to be running in the initial system state. Every active process is associated with a unique identifier number, which is stored

Listing 2.2 Process declarations

```
1: proctype A(byte p){
2:     printf("Hello, I'm process A and param = %d\n", p)
3: }
4:
5: active proctype B(){
6:     printf("Hello, I'm process B and my pid is %d!\n",_pid)
7: }
8:
9: init {
10:     run A(1); run A(2);
11: }
```

in the local, read-only variable `_pid`. The `init` keyword is used to declare an initial process. Such initial process cannot be parametric and can be instantiated only once. A process can instantiate new processes via the unary operator `run`, taking as argument a previously-declared `proctype`'s name and a possibly empty list of parameters matching the one in the `proctype` declaration, as shown in the `init` process in the example above. `printf`, as in the C programming language, is used for printing text during simulation runs. All the active processes execute concurrently in interleaving, i.e. only a single process can perform a computation step in a given instant of time.

Notice that, since Promela defines only finite state systems, the number of active processes is limited to 255.

Statements and expressions

In the previous examples in listings 2.1 and 2.2, basic statements like assignments and print statements have already been introduced. Other basic statements in Promela are send and receive statements over channels. Send statements are used to send a message over a channel and have the form `chName!args`. By default, a send statement is executable only if the channel is not full, but Spin also supports an alternative mechanism overriding the default behaviour and making send statements always executable. When using this alternative behaviour, messages sent to a full channel are lost. Receive statements are used to retrieve the first message in a channel and have the form `chName?args` or `chName?<args>`. A receive statement is executable iff the first message in channel `chName` matches the pattern in `args`. If a variable appears in the argument list, then the corresponding value of the message is copied into the variable when the message is received. If no square brackets were used, the first message is removed from the channel, otherwise there is no side-effect on the channel.

Expressions include, besides arithmetic and Boolean expression which are very similar to the ones in the C language, polling expressions over channels and run expression (an example of which is in Listing 2.2). Channel polling expressions are side-effect-free tests for the executability of receive statements and have the form `chName?[args]`. A channel poll evaluates to *true* iff the receive statement obtained from the poll expression

by removing the square brackets is executable.

Every expression in PROMELA is also a statement and is considered executable if its evaluation is equal to a non-zero value, i.e. it evaluates to *true*. If a process is at a control point where there is no executable statement to execute, it blocks.

Compound statements and control-flow elements

In PROMELA, a semicolon-separated list of statements enclosed in curly braces is called a *sequence* and treated syntactically as if it were a statement. Atomic sequences may be defined by prefixing a sequence with the keyword `atomic`. An atomic sequence is a sequence that is required to execute indivisibly and is not to be interleaved with other running processes, i.e. no other process is able to execute statements from the moment the first statement in the atomic sequence is executed until the last statement is completed. Notice that this is guaranteed only if all the statements in the atomic sequence are executable. If a blocking statement is encountered during the execution of the atomic sequence, atomicity is lost and other processes may execute. Once the blocking statement becomes executable again, the process must compete with other processes and be scheduled to execute before the execution of the atomic sequence can be resumed.

Selection construct The selection construct is used to define a choice between the execution of multiple options. A selection construct is started by the keyword `if`, ended by `fi` and contains at least one *option sequence*. Each option sequence starts with a double-colon and can be selected only if its first statement (called its guard) is executable. The whole selection construct is executable if at least one of its options' guards is executable. The predefined condition statement `else`, executable if and only if no other statement is executable for the process in the current control state, may be used as a guard for at most one option sequence in the selection construct. If more than one guard is executable, then the option sequence to execute is chosen non-deterministically from those having an executable guard. After executing an option sequence, the process moves to the control state following the selection construct. Listing 2.3 shows a selection construct. Note that the arrow '`->`' is a statement separator and is equivalent to '`;`'.

Repetition construct A repetition construct is started by the keyword `do`, ended by `od` and, much like the previously-described selection construct, contains at least one *option sequence*, each starting with a double-colon. In each iteration, an option sequence can be

Listing 2.3 Selection construct example

```
1: byte a=10,b=11;
2: if
3: :: (a<=10) -> printf("first option"); a=0;
4: :: (b==11); printf("second option"); b=0;
5: :: else -> printf("else case")
6: fi
```

Listing 2.4 Repetition construct

```
1: short a,b;
2: init {
3:     do
4:     :: (1)    -> a++; b++
5:     :: (1)    -> a--; b--
6:     :: (a>0) -> break;
7:     od
8: }
```

Listing 2.5 Unless construct

```
1: short a,b;
2: init {
3:     do
4:     :: (1) -> a++; b++
5:     :: (1) -> a--; b--
6:     od unless {(a>0)}
7:
8: }
```

selected only if its first statement (called its guard) is executable and, if more than one option sequence can be selected, the option to execute is selected non-deterministically. After executing an option, the control flow returns to the repetition construct. The `break` keyword causes the control flow to move to the statement immediately after the repetition statement. Listing 2.4 show an example of repetition construct. In the example, the first two options are always executable. If the variable `a` is positive, all three option sequences can be non-deterministically selected. The loop terminates when the last option is selected. The predefined, global, read-only, Boolean variable `timeout` that is true in all global deadlock states, i.e. in those global states in which no process has executable statements, can also be used as a guard in option sequences in repetition and selection constructs.

Unless construct An unless construct has the form `main unless escape`, where `main` and `escape` are blocks or a sequences of statements and are respectively called *main sequence* and *escape sequence*. The executability of all statements inside the main sequence is constrained to the non-executability of all guard statements in the escape sequence. If a guard in the escape sequence becomes executable, the execution continues with the remainder of the escape sequence. If no guards in the escape sequence becomes executable during the execution of the main sequence, the escape sequence is skipped and execution continues from the following state. An example of unless construct is shown in 2.5. Notice that this example is not equivalent to the repetition construct in 2.4: in the latter it is possible to choose the third option and leave the repetition only during option selection phase, while with the `unless` any statement is interrupted as soon as the guard in the escape sequence becomes executable.

`goto` statements In Promela, control states can be labelled by prefixing statements with a label name which must be unique within the surrounding `proctype` or `never` claim. A statement of the form `goto L` moves the control state to the one named `L`. For examples of labelled control states and `goto` statements see Listing 2.6.

Never claims

A `never` claim is used to define a finite or infinite system behaviour that is of special interest, usually because it should never occur. Never claims may be hand-written or generated by Spin starting from a temporal logic formula [19]. A finite behaviour is matched if the claim reaches its final state, i.e. the control arrives at the closing curly brace. Infinite behaviour is matched if the claim visits an accepting state infinitely often. Since Promela models have a finite number of states, an infinite behaviour necessarily results in a path leading to a cycle. In particular, infinite behaviours visiting an accepting state infinitely often result in a path leading to a cycle containing an accepting state. Thus, to check whether an infinite behaviour matches, the verifier looks for a reachable cycle containing a control location labelled with an accepting label, namely a label starting with `accept`, e.g. `accept_all`, `acceptstate`, etc. Spin translates the never claim to a different process and executes it in lockstep with the `proctypes`, i.e. each global system step can be seen as two transitions: one from the never claim process and the other from one of the other processes, with the never claim process always moving first. If no statement is executable in the never claim process, then no further move is possible in the current path and a new execution is explored [21]. In the examples in Listing 2.6, the `never` claim `n1` matches the finite behaviours in which the condition `(State==S1)` is eventually satisfied. The `skip` statement used in line 8 is a predefined dummy, always-executable, statement having no side effects and is used because blocks cannot terminate with a label. In `n1`, execution starts from the `never_step` selection statement. Since there is an option with the `else` guard, the selection statement is always executable. If the condition `(State==S1)` is satisfied the control passes to the `end_never skip` statement and, after executing the skip, reaches the final state and matches the never claim. If the condition is not satisfied, control returns to the selection statement. The never claim `n2` matches infinite behaviours in which the condition `(State==S1)` holds infinitely often. To do so, it is necessary to define an accepting state `accept`. Execution in `n2` starts in `T0_init` and enters the repetition construct. Then, in each iteration, if the condition is satisfied, the control passes to the `accept` state, otherwise, control remains in `T0_init`. Once the `accept` state is reached, control returns to `T0_init`. If a behaviour causes the never process to visit infinitely often the `accept` state, then in said behaviour the condition `(State==S1)` holds infinitely often.

2.2 Deriving Promela models from DSTMs

This section describes a fully-automatable procedure to translate a DSTM model into a Promela specification. An initial version of this translation process was originally presented in [5, 6] and fully implemented – part in Java and part in the Atlas transformation language – within the context of the CRYSTAL project.

This first version, as made clear by the thorough analysis carried out as part of this thesis work, fails to faithfully adhere to the DSTM formal semantics and is afflicted with several, mainly concurrency-related, issues. An important part of this thesis work con-

Listing 2.6 Never claims

```
never n1 {
    never_step:
        if
        :: (State==S1) -> goto end_never
        :: else -> goto never_step
        fi;
    end_never:
        skip
}
never n2 {     /* []<>(State==S1) */
    T0_init:
        do
        :: (State==S1) -> goto accept
        :: else -> skip
        od;
    accept: /* accepting control state */
        do
        :: (1) -> goto T0_init
        od;
}
```

sisted in finding ways to address these issues and in patching the existing implementation accordingly. In what follows, an updated version of the translation process is described and particular emphasis is put on the original contributions of this thesis work.

2.2.1 An overview of the translation process

The proposed translation is a two-step process. The first flattening step transforms the given DSTM model into ordinary state machines by removing the hierarchical structure. This flattening step is necessary since Promela has no support for hierarchical specifications. The second step transforms the resulting ordinary state machines into an actual Promela specification which also takes care of modelling a possibly non-deterministic environment. These two steps are described in detail in the following sections 2.3 and 2.4, respectively.

2.3 Flattening the DSTM into ordinary state machines

This subsection describes a procedure to transform a hierarchical DSTM specification into a sequence of concurrent ordinary (flat) state machines, which are easily encodable in Promela.

In order to remove hierarchical structure from a DSTM model, it is necessary to remove all boxes, forks and joins and substitute them with suitably-defined nodes and transitions, which are also used to model the activation of other flat machines and termination synchronization.

When removing a box, three different situations are possible, depending on the structure of the DSTM model. Therefore, three different mapping schemata are defined:

simple box all the transitions entering the box have as source other boxes or nodes;

asynchronous fork the source of the transition entering the box is a fork pseudo-node and there exists an asynchronous fork transition from the fork pseudo-node to a node. Note that, by Definition 2, in a well-formed DSTM, if a transition from a fork enters a box, then no other transitions are allowed to have the box as target.

synchronous fork the source of the transition entering the box is a fork pseudo-node and there exists no asynchronous fork transition exiting the fork pseudo-node.

In the following, each mapping schema is described in detail.

Simple box

In this case the box is substituted by a node having the same name. All transitions whose source (resp. target) is the box are substituted by transitions exiting (resp. entering) the newly-created node and inherit the original transitions decorations. Those decorations, however, must be suitably enriched in order to model the instantiation of the other machines associated with the box and allow the calling machine to notice the called-machines eventual termination. When dealing with entering transitions, triggers and guards stay unchanged, but it is necessary to add more actions to account for the instantiation of the machines associated with the box. Hence, for each of such machines, a *run* action is added, corresponding to the Promela `run` instruction performing process activation. A run action has the form `run MachineName(paramList)`. `paramList` is a list associating to each parameter in the called machine a term in its domain and contains also additional parameters, needed to correctly handle machine termination and the steps semantics. In particular, the following additional parameters are added:

- a `parent` parameter, containing the process identifier of the process being above the one being instantiated in the hierarchy. In the *simple box* case, this parameter corresponds to the calling process' identifier;
- an `initialState` parameter, used to specify which entering state is the initial state for the instance being instantiated;
- for a machine having n exiting states, $n + 1$ parameters are added, each to be actualized with an internal channel name. These channels are used to handle process termination. In particular, for each exiting node `exit` in the machine `MachineName` instantiated by the box `boxName`, a channel of `bits chTerm_boxName_MachineName_exit` is introduced. Each of these channels is used by the called process to signal the reaching of the corresponding exiting state to the calling process. An additional channel of `bits` named `chTerm_boxName_MachineName` is used by the caller process to issue a termination message to the called process.

As regards the transitions exiting the box, each of them is replaced by a transition exiting the node substituting the box and inheriting the original transition's decoration. If the original transition is a *return by exiting*, the guard needs to be enriched with a new condition checking for the termination of the instantiated machine in the required state having the form `chTerm_boxName_MachineName_exit[?<1>]`. If the original transition is a *return by default*, the guard needs to be enriched with a new condition checking for the termination of each of the called machines. To check for a machine termination, regardless of the exiting state, one can simply use a disjunction of all conditions of the form `chTerm_boxName_MachineName_exit[?<1>]`, one for each exiting node `exit`. So, the new condition to be added is a conjunction of machine termination checks, one for each machine instantiated by the box. If the original transition is a *return by interrupt*, there is no need to enrich the transition guard. Moreover, in either case, when a return transition fires all the called processes must terminate. This is achieved by adding, for each called process, an action of the form `chTerm_boxName_MachineName!<1>` sending a termination message to the called process and an action of the form `chTerm_boxName_MachineName_exit!<_>` for each exiting node `exit` to clean the corresponding channel from the eventual termination messages.

Also notice that the proposed naming schema for termination-handling channels is intended to be a simplistic example for explanation purposes and may lead to ambiguity in particular cases, e.g. when a box instantiates two instances of the same machine, when two boxes in different concurrent machines have the same name and instantiate the same machine, or when processes are dynamically instantiated via asynchronous forks. These ambiguities shall be addressed when implementing the transformation. For the level of detail of this description it suffices to assume that a distinct communication channels is used for each *caller-called* couple of processes and for each *caller-called exiting node* couple.

Asynchronous fork

After performing an asynchronous fork the calling process continues to run concurrently with the newly-instantiated processes. By well-formedness (see Definition 2), if there is an asynchronous fork transition then there is also a corresponding join pseudo-node. Moreover, all the control flow between the fork and the join is contained within the boxes entered by the fork, i.e. the fork enters a set of boxes and the return transitions from these boxes must enter the associated join. Hence the fork, the join and the boxes entered by the fork can be considered as a single block to be removed and substituted with suitable transitions. Some of these transitions model the fork operation and lead from the source node of the *entering fork* transition to the target node of the *asynchronous fork* transition. These transitions instantiate the necessary processes with appropriate *run* actions and also take care of modelling the different permutations of the actions associated with the transitions exiting the fork pseudo-node (see Definition 10). Notice that, in this case, the `parent` parameter corresponds to the parent of the calling process, as the new processes instantiated by an asynchronous fork are siblings of the calling

process in the hierarchy tree. In a similar way, to model the join operation it is necessary to add one or more transitions, depending on the join being preemptive or not. Each of these transitions leads from the target node of the asynchronous fork transition to the target of the exiting join transition. If the join is non-preemptive, it is replaced by a single transition inheriting trigger and guard from the *entering join* transition and enriching the guard with a condition requiring the termination of each machine instantiated by the joined boxes. Moreover, the action needs to deal with the termination of the spawned processes – which is done as described in the *simple box* case above – and must include the action of the exiting join transition. If the join is preemptive, it is replaced by a set of transitions, one for each transition entering the join and qualified as preemptive. Each of these transitions inherits the same trigger and guard as the original preemptive transition they are associated with. If the original preemptive transition is a *return* (either by default or by exiting) having source in a box b, the guard needs to be enriched with appropriate conditions requiring the termination of the machines associated with b as previously described in the *simple box* case. The actions are defined as in the non-preemptive case.

Synchronous fork

In the synchronous fork case, the calling process suspends itself and waits for the termination of the called processes. In this case, the fork pseudo-node, the boxes the fork enters and the (optional) associated join are considered as a single block and are replaced by a new `wait` node and suitably defined transitions to and from the newly-introduced `wait` node. The new transitions introduced to model the fork operations are defined as in the *asynchronous fork* case, with the exception of the `parent` parameter receiving in this case, as in the *simple box* one, the process identifier of the calling process. The transitions necessary to model the join operations are defined as in the *asynchronous fork* case.

Example 8 (Flattening the *Counting* DSTM specification)**.** Consider the *Counting* DSTM specification detailed in Figure 1.1 and in Table 1.1. Its flattening is represented in Figure 2.2 and detailed as follows. When flattening the Main machine, the box counterBox needs to be removed and falls in the *simple box* case. So, counterBox is removed and replaced by a node having the same name. Transition T2 is replaced by a transition having the same name and inheriting its decoration, with an additional *run* action of the form `run Counter(100,pid,default,...)`, where `100` is the actualization for the `P_to` parameter, `pid` is the calling process identifier, `default` is the initial state for the Counter machine, and additional parameters are the channels `chTerm_counterBox_Counter_limit`, and `chTerm_counterBox_Counter`. T3, being a *return by interrupt* transition, is replaced by a transition with the same name exiting the node counterBox and inheriting the original decoration, with additional actions having the form `chTerm_counterBox_Counter!<1>` and `chTerm_counterBox_Counter_limit?<_>`, used to send a termination signal to the spawned Counter process and to remove the received termination signal from the channel. T4, being a *return by exiting* transition, is replaced by a transition with the same name exiting the node counterBox and inheriting the original decoration. Its guard is

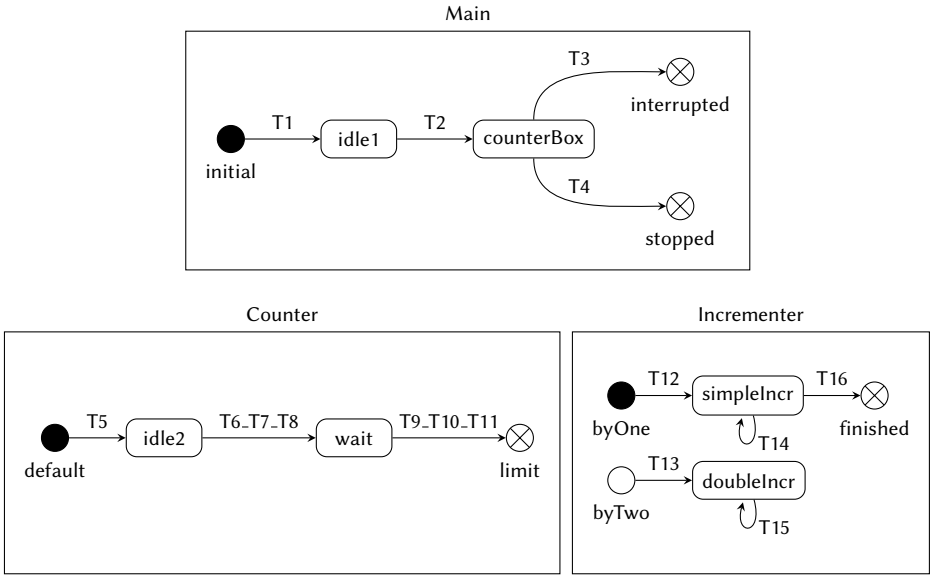


Figure 2.2: The flattened *Counting* DSTM specification

enriched with an additional condition requiring for the termination of the Counter machine (`chTerm_counterBox_Counter_limit[?<1>]`) and an additional action sending a termination signal to the spawned Counter process is added, as in T3.

When flattening the Counter machine, the fork, the join and the boxes in between need to be removed as detailed in the *synchronous fork* case. Therefore, the fork, the join and the boxes are removed and replaced by a wait node and transitions T6_T7_T8 and T9_T10_T11. T6_T7_T8 models the fork operations and inherits trigger and guard from the *entering fork* transition T6 (recall that, as detailed in Table 1.2, call transitions having as source a join pseudo-node must have trivial triggers and guards). Two additional *run* actions having the form `run Incrementer(P_to,pid,<init>,...)` are introduced, with `P_to` being the actualization for Incrementer's parameter `P_limit`, `pid` being the process identifier of the calling process, `<init>` being respectively byOne and byTwo (as one of the call transitions exiting the fork is a *call by entering*), and additional parameters being `chTerm_<box>_Incrementer_finished` and `chTerm_<box>_Incrementer`, with `<box>` being respectively boxIncr1 and boxIncr2. Since there is no action associated with the call transitions T7, T8, a single transition suffices to handle all possible permutations. As the join is preemptive and T9 is the only transition entering the join and qualified as preemptive, the join operations are modelled by the single transition T9_T10_T11 leading from the wait node to the exiting limit node, target of the *exiting join* transition T11. Such action has a guard requiring the termination of the Incrementer machine instantiated by boxIncr1 (`chTerm_boxIncr1_Incrementer_finished[?<1>]`), inherits its action from T11 and has additional actions needed to send the termination signal to each joined machine (`chTerm_<box>_Incrementer!<1>`, with `<box>` being in {boxIncr1, boxIncr2}) and to remove the message from channel `chTerm_boxIncr1_Incrementer_finished`. The Incrementer machine is unaffected by the flattening step since it contains neither

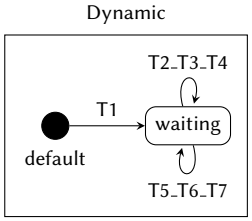


Figure 2.3: The flattened *Dynamic* DSTM specification

boxes nor forks.

Example 9 (Flattening the *Dynamic* DSTM specification)**.** As an example of flattening involving an asynchronous fork, consider the *Dynamic* DSTM specification detailed in Figure 1.2 and in Table 1.3. Figure 2.3 depicts the flattening of the *Dynamic* DSTM model, as explained in what follows. The entire block containing the fork, the join, and the boxIncr box is replaced by the two transitions T2_T3_T4 and T5_T6_T7, modelling respectively the fork and the join operations. Transition T2_T3_T4 leads from the source of the *entering fork* transition T2 to the target of the *asynchronous fork* transition, and is therefore a self-loop on the waiting state. The transition inherits triggers and guards from the *entering fork* transition T2 and, in addition to the actions inherited from transitions T2, T3 and T4, an appropriate run action is added to instantiate the Incrementer machine, similarly to what shown in the previous example. No other transition is necessary to model the fork as T3 is decorated with the empty action and there is only one possible permutation for the actions exiting the fork. T5_T6_T7 models the join operation and leads from the target node of the asynchronous fork transition to the target of the exiting join transition, therefore it is –again– a self-loop on node waiting. This transition inherits trigger and guard from the entering join transition T2, has an additional guard condition requiring the termination of the Incrementer machine (`chTerm_boxIncr_Incrementer_limit[?<1>]`), inherits the action from T7 and has the usual additional actions to send the termination signal to the Incrementer machine and to remove the termination message from channel `chTerm_boxIncr_Incrementer_limit`. As already noticed in the previous example, the Incrementer machine is not affected by the flattening step as it does not contain any box or fork.

2.4 Promela encoding for the flat model

To obtain a procedure to translate a flattened model into a Promela specification it is necessary to address the following key points:

- translation of data-flow elements;
- encoding of flat machines;

Listing 2.7 Multi-type and external channel mapping in Promela

```
1: mtype={white,red,green};
2: chan ch_Int = [3] of {bit, int};
3: chan ch_Colour = [3] of {bit, mtype};
4: chan chExt = [2] of {bit,int} /* external channel */
```

- orchestration of the concurrent flat machines and correct realization of the steps semantics.

This section deals first with the translation of data-flow elements, i.e. how to map a DSTM type, variable or channel to its Promela equivalent, in Subsection 2.4.1. Then, after an overview of the Promela specification's structure in 2.4.2, Subsection 2.4.3 deals with the simplified problem of mapping non-hierarchical DSTM models to Promela. Subsection 2.4.4 presents a mechanism to correctly enforce DSTM steps semantics and, finally, Subsection 2.4.5 provides a general schema to map DSTM models to Promela specifications.

2.4.1 Translation of data-flow elements

The mapping of DSTM types and variables to their Promela equivalent is rather straightforward, with the DSTM type **Int** being naturally mapped to the Promela `int` type DSTM user-defined enumeration basic types being mapped to `mtypes` and compound types being mapped to suitable user-defined Promela datatypes declared with `typedef`.

Internal DSTM channels are mapped to Promela channels having a buffer size equal to the bound of the DSTM channel. If the DSTM channel is not a multi-type channel, the type of the messages conveyed by the Promela channel is obtained from the DSTM channel type as described above. Conversely, if the DSTM channel is a multi-type channel, it is modelled by a set of Promela channels, one for each simple type. These channels are managed in a way that guarantees that, in each position, at most one of them contains a valid message. This can be achieved by adding a validity `bit` field to each message in the channels. As an example, consider the basic DSTM enumeration type `Colour` = {*white*, *red*, *green*} and the multi-type `MT` = {**Int**, `Colour`}. The DSTM internal channel *ch*, having $bd(ch) = 3$ and $type(\hat{ch}) =$ `MT`, is mapped to the two Promela channels `ch_Int`, `ch_Colour` as shown in Listing 2.7.

External channels are encoded in Promela by channels having buffer size of two, with the first position containing the message for the current step and the second containing the eventual message for the next step, if produced during the current step. In order to ensure that messages produced in the current step are stored in the second position and avoid that a message produced during the current step triggers transitions in other processes in the same step, the external channels are managed in a way that guarantees that the first position is always filled. To do so – as with the multi-type internal channel mapping – an additional validity `bit` field is introduced in every message, so that an empty external channel can be modelled by a channel containing an invalid message in

Listing 2.8 External channel management code

```
1: chan ext = [2] of {bit,bit}; //external channel declaration
2: if
3: :: (len(ext) < 2) -> { // must generate new message
4:     if
5:     :: (1) -> {ext!0,0} // invalid message
6:     :: (1) -> {ext!1,0} // valid message
7:     :: (1) -> {ext!1,1} // valid message
8:     fi
9:    }
10: :: else -> skip //system already generated next step message
11: fi
12: if
13: :: len(ext)==2 -> ext?temp1,temp2
14: :: else -> skip //len(ext)=1 on the first step
15: fi
```

the first position. As an example, consider the external channel *chExt* of type `Int`. Its Promela mapping is the channel `chExt` shown in Listing 2.7. To comply with the DSTM specification, additional management operations on external channels are required. In the proposed encoding, such actions are assigned to the mentioned *Engine* process. In what follows, the necessary management operation for external channels are described. At the beginning of the initial step (see Section 1.3.3) a possibly empty message for each external channel is generated non-deterministically and sent in the first position. If an empty message is generated, an invalid message of the form `(0,_)` is sent. Before each subsequent step, the first message is removed from every external channel and, after that, if the external channel is empty – i.e. there was no message generated during the previous step in the second position – a possibly empty message is generated non-deterministically as in the previous case. An example of Promela code to perform said management operation on the external channel `chan ext = [2] of {bit,bit}` is shown in Listing 2.8. The selection construct in lines 2–11 has two option sequences. The first one (line 3) is executable when the external channel is not full, i.e. when the next step is the first one or when, during the previous step, no external message was generated by the system. In this case a possibly-invalid new message is non-deterministically generated (lines 5–7) and sent over the channel. The second option (line 10) is executable only when the channel is full, i.e. the system has generated a message during the previous step, in which case no message generation occurs. Then control passes to the second selection construct (lines 12–15) in which, if the channel is full, the first message is consumed, otherwise the dummy statement `skip` is executed. This selection is necessary to avoid consuming the channel's message before the first step, when only a message is present.

2.4.2 An overview of the Promela specification

Given a DSTM model $D = \langle M_1, \ldots, M_n, X, C, P \rangle$, the proposed Promela encoding is structured as follows: (i) an initial section containing global declarations of datatypes,

variables, and channels; (ii) an `active proctype` named `Engine`, the purpose of which is to initialize external channels before each step and to ensure that the DSTM steps semantics is fulfilled; (iii) a `proctype` declaration for each of the n machines;

2.4.3 Mapping a flat DSTM to a Promela specification

Before addressing the more complex issues related to process synchronization and steps semantics, this subsection deals with the simpler problem of Promela-encoding a DSTM specification containing a single flat machine. This simplified construction shows the main intuitions behind the proposed translation process and will be properly extended to work in the general case in the following subsections.

Consider a flat DSTM $F = \langle M_1, X, C, P \rangle$. The flattening step does not alter the structure of M_1, since it is already a flat machine, and only adds parameters and enriches transitions decorations as described in 2.3. The general schema shown in Listing 2.9 can be used to obtain a Promela encoding from the flattening of F. In the first part, global variables, channels and datatypes used in the DSTM model F are declared in Promela, as previously described. Moreover, symbolic constants to refer to states are introduced in line 5. The `HasToken bit` array declared in line 7 is needed to avoid sequential firings of transitions within the same step, as will be soon explained.

Starting from line 9, the `proctype` for the flattened machine M_1 is defined. The `M1 proctype` has the same parameters described in 2.3 and declares a local `mtype` variable named `state` used to store the current state, initializing it to the value received in the `initial` parameter. After that, the process enters the main repetition construct, exiting only when a termination signal is received on the dedicated channel `chTerm` (see the `unless` construct at line 24). Inside the main repetition construct there is one option sequence for each machine state `S`, having guard `(state == S && HasToken[_pid]==1)`. Each option sequence immediately consumes the token (see line 16) and then enters a selection construct to (possibly non-deterministically) choose which transition to perform. Such transition selection construct contains an option sequence for each transition t exiting the state `S`. Each option sequence is guarded by a condition having the form $(\xi \wedge \phi)$, with ξ and ϕ being respectively the trigger and the guard associated with the transition, and performs the required actions specified in the transition's decoration. Moreover, each option sequence takes care of setting the `state` variable to the corresponding transition's target.

Starting from line 30, the `Engine` process is defined and qualified as `active`. After declaring and initializing the channels required for termination synchronization with `M1` (line 33), `Engine` activates an instance of the `M1 proctype` and enters a repetition construct with three option sequences: the first (line 36) is enabled when a termination signal from `M1` is received on the dedicated channel and, after sending a termination signal back to `M1`, allows `Engine` itself to exit the repetition construct and terminate; the second is enabled only when no other process has enabled statements and at least one transition has been performed during the current step, and, after performing the necessary management operations on external channels, resets the `HasFired` flag for

Listing 2.9 Flat DSTM encoding schema

```
#define MAX_PROC 2; // maximum number of concurrent processes
// Global variables, channels, datatypes declarations
bit x; int y; chan c = [8] of bit; ...
// Mtype declarations for each machine's state name
mtype = {S1, S2, S3, ...};
bit HasFired; // set to 1 if a trans. fired during current step
bit HasToken[MAX_PROC]; // track which processes own the token

proctype M1(pid parent,mtype initial,chan chTerm,chan chTerm_ex){
  mtype state = initial;
  do // main repetition construct
  // for each state S ∈ N_i ∪ En_i
  :: (state == S && HasToken[_pid]==1) ->
    atomic {
      HasToken[_pid]==0; // consumes token
      if // transition selection construct
      // for each trans. t with Src_1(t) = S, Trg_1(t) = T, Dec_1(t) = ⟨ξ, φ, α⟩
      :: (ξ ∧ φ) -> {
          α; state = T; // sets next state
          HasFired = 1;
        }
      fi
    }
  od unless {
    (chTerm?[1]) -> // if termination signal arrives
      chTerm?1;
  }
}

active proctype Engine(){
  pid PidMain;
  // initialize channels required for termination sync with M1
  chan chT_M1 = [1] of bit, chT_M1_exit = [1] of bit;
  PidMain = run M1(_pid,init,chT_M1, chT_M1_exit);
  do
  :: (chT_M1_exit?[1]) -> {chT_M1!1; break}
  :: (timeout && HasFired == 1) -> {
      // initialize external channels for next step
      HasFired = 0; HasToken[PidMain] = 1
    }
  :: (timeout && HasFired == 0) -> {chT_M1!1; break}
  od
}
```

the next step and passes the token again to the `M1` process; the third is enabled when no other process has executable statements and no transition fired during the current step and, much like the first option sequence, makes the `Engine` terminate after sending a termination signal to `M1`.

During a normal execution step, process `M1` receives the token from `Engine` and this makes the main repetition construct executable for `M1`, since one of the option sequences is necessarily enabled. After entering the option sequence corresponding to its current state, `M1` consumes its token and enters the transition selection construct. Here, assuming that the selection construct is executable – i.e. there is at least one enabled transition – `M1` selects an option sequence, executes it and returns to the main repetition construct, which will now necessarily be non-executable since all sequence options are guarded by an `HasToken[_pid]` condition. The unavoidable deadlock makes the second option in the repetition construct in `Engine` executable and the next step can start. Without the `HasToken` condition, `M1` could fire multiple transitions in a single step without the proper external channel management operations being performed by `Engine`. On the contrary, suppose that the transition selection construct is not executable during a certain step, then `M1` just blocks without setting the `HasFired` flag and the third option in the `Engine` repetition construct becomes executable. In this case, the current execution has "wasted" a step (perhaps because the external signals required to make the system progress were not generated during the non-deterministic signal generation). In this case there is no need to continue the current run, since there will necessarily be a "luckier" run where the required signals are generated earlier and the same behaviour will occur without wasting steps.

2.4.4 Enforcing the steps semantics

Accordingly to the DSTM semantics, given a global system state $s = \langle\langle Vx, \lambda\rangle, Fr, \theta\rangle$, the machine M_i is allowed to execute and perform a compound transition ct iff the compound transition is enabled with regard to the state s and to a set N of children of M_i in the control tree, i.e. the predicate $Enabled(ct, N)$, as defined in 1.3.3, holds in s. With the removal of explicit hierarchy during the flattening step, there are no compound transitions anymore and the above conditions can be simplified. A process is allowed to execute and perform a transition iff:

(i) it has not executed yet during the current step (sequential firing of transitions is forbidden);

(ii) none of the processes it instantiated and of their descendants has executed;

(iii) none of its ancestors can perform a transition.

Conditions (i) and (ii) are enforced in the DSTM semantics by the frontier itself, with the nodes being source of a transition in a step being required to belong to the frontier and being removed from the frontier along with their descendants during the transition

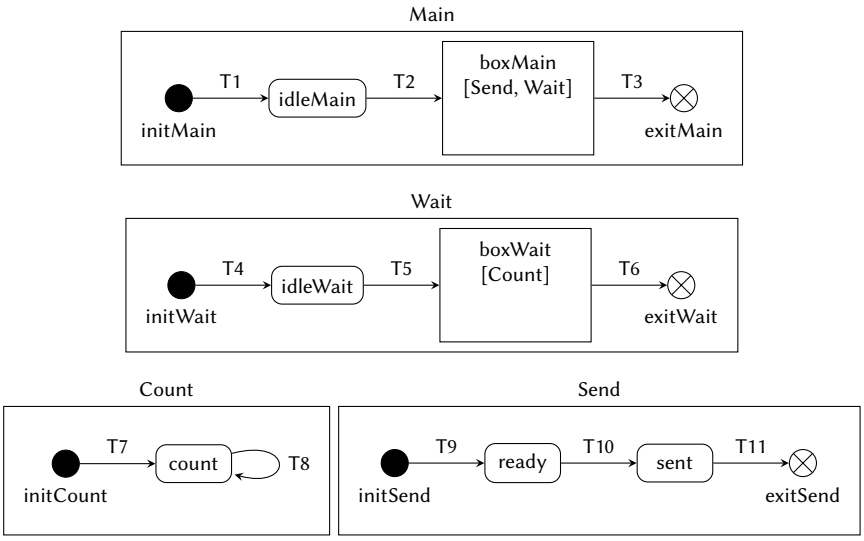


Figure 2.4: The *Prelation* DSTM model

firing (see definition 10). Condition (iii) comes from the definition of enabled-ness for compound transitions given in 1.3.3.

In the initial version of this translation process [5, §5.2], Benerecetti et al. dealt with the enforcement of steps semantics with the *token-passing* mechanism described as follows. Each PROMELA process, modelling a DSTM machine, is required to own a token in order to perform a transition. When a process holding its token is scheduled, it consumes its token and tries to execute a transition. If no transition is executable, the process passes its token on to its children. When no process is able to execute, because every process either has consumed its token or has never received it from its parent, the *Engine* process starts a new step. At the beginning of each step, the *Engine* process passes the token back to the main machine.

This mechanism is however incorrect and could lead to non-maximal steps, as shown in Example 10.

Example 10 (Token-passing mechanism failure)**.** Consider the DSTM model

$$Prelation = \langle M_1, M_2, M_3, M_4, X, C, P \rangle,$$

where $X = \{x\}$, $C = \{\texttt{sig}\}$, $P = \emptyset$, and M_1, M_2, M_3, M_4 are respectively the machines Main, Wait, Count, and Send depicted in Figure 2.4 and detailed in Table 2.2.

Suppose that *Prelation* has been flattened as described in 2.3 and that a PROMELA encoding has been produced. Consider now the step in which the Send machine sends the signal over channel `sig`. At the beginning of the step, *Engine* passes the token to the Main machine, which is in the state corresponding to the box boxMain and has no enabled transitions. Having no enabled transitions, Main passes the token to its children Wait and Send. Suppose that the Wait process is scheduled before Send. Wait has no enabled transitions, since transition T6 requires a message on channel `sig`, and passes its

T_1	Src_1	Trg_1	Dec_1	$Inst_1$
T1	initMain	idleMain	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T2	idleMain	boxMain	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T3	boxMain	exitMain	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T_2	Src_2	Trg_2	Dec_2	$Inst_2$
T4	initWait	idleWait	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T5	idleWait	boxWait	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T6	boxWait	exitWait	$\langle$ `sig?1`, $True, \varepsilon \rangle$	$\varnothing$
T_3	Src_3	Trg_3	Dec_3	$Inst_3$
T7	initCount	count	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T8	count	count	$\langle \tau,$ `x<10`, `x=x+1` $\rangle$	$\varnothing$
T_4	Src_4	Trg_4	Dec_4	$Inst_4$
T9	initSend	ready	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$
T10	ready	sent	$\langle \tau,$ `x>=10`, `sig!1` $\rangle$	$\varnothing$
T11	sent	exitSend	$\langle \tau, True, \varepsilon \rangle$	$\varnothing$

Table 2.2: Transition structure for the *Prelation* DSTM model

token to its child Count, which likewise cannot perform any transition and just consumes its token. When the Send process is selected by the non-deterministic Spin scheduler, it executes transition T10 and sends a message on channel `sig`. After that, no other process can execute since no one owns the required token and the step abruptly ends with a non-maximal set of transitions, as transition T6 is now enabled and should be executed in the current step.

The main intuition behind the new mechanism devised during this thesis work to correctly enforce steps semantics is to perform more than one token descent during the simulation of a DSTM step. Similarly to the previously-described mechanism, at the beginning of each step the main process and possibly its siblings in the process hierarchy – if the main process performed asynchronous forks – receive the token from *Engine*. Every process owning the token consumes it when scheduled and tries to execute a transition. If no transition is executable, the process passes its token on to its children. This token-passing process continues until reaching a state in which no process is able to execute, because every process either has consumed its token or has never received it from its ancestors which have executed. At this point, the mechanism described in [5] would conclude the current step and prepare the system for the next one. On the contrary, in the new mechanism, *Engine* passes the token again to the main process and to its siblings until a deadlock is reached without any transition being fired during the token descent phase. When this happens, the current step ends and *Engine* can initialize the next one. With the new mechanism admitting multiple token descents, a process whose child executed in a previous descent might be able to execute within the same step, thus violating DSTM semantics. To address this issue, a new flag `DescendantExecuted` is introduced and set to

true for all those processes having a descendant which already executed during the current step. After the execution token is propagated downwards in the process hierarchy, the `DescendantExecuted` flag is propagated upwards from those processes which directly performed a transition to their parent, until the root of the hierarchy is reached. This is achieved by using a fictitious `backProp` state, as explained below. The process of passing the token to the main process and its sibling, propagating the token downwards the process hierarchy and subsequently propagating the `DescendantExecuted` flag upwards from those processes who directly performed a transition to their ancestors is called a *phase*. Once a *phase* is concluded, *Engine* can either start a new phase by giving the token again to the main process and to its siblings, or start a new step, if no transition was fired during the latest phase. Clearly, the conditions guarding the execution of a transition for a process need to be suitably enriched considering the new `DescendantExecuted` flag, allowing only those processes owning the token and not having the `DescendantExecuted` flag set to perform transitions. The proposed mechanism is described in greater detail as follows:

(i) at the beginning of the step, after the proper management operation for external channels are performed, *Engine* passes the token to the main process and to its siblings. At this point, every process has its `DescendantExecuted` flag set to false. The global flags `HasFired`, `isFirstDescent`, and `updateState` are set to false;

(ii) every process owning the token and not having its `DescendantExecuted` flag set, consumes its token when scheduled and tries to execute a transition. If a transition is executed, the global flag `HasFired` is set to true and a local variable `nextState` is used to store the machine's next state. If no transition is executable, the process passes its token on to its children and `nextState` is set to hold the same value as `state`. In either case, `state` is set to the fictitious state `backProp`, entering the backpropagation sub-phase in step (iii);

(iii) every process in the `backProp` state can execute if its `DescendantExecuted` flag is set, in which case it sets the `DescendantExecuted` flag for its parent. Once a deadlock state is reached, execution continues with step (iv);

(iv) *Engine* activates and

  (a) if the `HasFired` flag is set and the `updateState` is not, then the current descent is completed and its necessary to reset the `state` variable for each process from the `backProp` state to its intended value stored in `nextState`. This is done by setting the global `updateState` flag and continuing as in step (v);

  (b) if the `HasFired` and the `updateState` flags are set, a new descent is to be performed within the same step. The `isFirstDescent` and `hasFired` flags are unset, the main process and its sibling receive the token again, and execution continues from step (ii);

(c) if the `HasFired`, `isFirstDescent` and `updateState` flag are unset, then execution continues like in (iv).(a);

(d) if the `HasFired`, `isFirstDescent` are unset and the `updateState` flag is set, the current step is concluded and execution continues as in (i);

(e) if the `HasFired` flag is unset and the `isFirstDescent` is set, then the current computation has "wasted" a step and is aborted, for the same reasons already discussed in 2.4.3.

(v) every process being in the `backProp` state is allowed to execute when the global flag `updateState` is set. During this sub-phase, each process resets its state to the intended value stored in `nextState`. Once every process has been scheduled and has restored its state, a deadlock occurs and execution continues from step (iv).

To better explain this rather complex mechanism, Example 11 shows how it behaves in the same situation described in Example 10.

Example 11 (The new mechanism in action)**.** Consider again the *Prelation* DSTM model described in Example 10. Suppose that it has been flattened as described in Section 2.3, that a PROMELA encoding has been produced, and consider – like in the previous example – the step in which the Send machine sends the message over channel `sig`. At the beginning of the step, *Engine* performs the proper initializations as described in (i) and passes the token to its child Main, which is blocked in its boxMain state, since neither the Send nor the Wait machine have terminated. Having no executable transition, Main sets its `nextState` to `boxMain`, its `state` to `backProp`, and passes the token on to its children Wait and Send. Suppose that Wait is scheduled before Send. Wait, being in the boxWait state, has no executable transition, thus sets its `nextState` to `boxWait`, its `state` to `backProp`, and passes its token to its child Count which is in the count state and, similarly, cannot perform any transition, sets its `nextState` to `count`, its `state` to `backProp`, and consumes its token. When the Send process is scheduled, it executes transition T10, sets its `nextState` to `sent`, its state to `backProp`, its `DescendantExecuted` flag and the global `HasFired` flag to true, and consumes its token. At this point, process Send can execute again, because its `state` is `backProp` and its `DescendantExecuted` flag is set (see point (iii) in the previous description), thus it sets its parent's (Main) `DescendantExecuted` flag. After that, Main similarly sets *Engine*'s `DescendantExecuted` flag and the system reaches a deadlock state. *Engine* regains control (see point (iv)) and, since the `HasFired` flag is set and the `updateState` is not, proceeds as described in (iv).(a). At this point (see point (v) in the previous description) every process restores its `state` from `backProp` to the value stored in `nextStep`. After that, the system finds itself in deadlock again and control returns to the *Engine* process. This time, both the `HasFired` and the `UpdateState` flags are set and *Engine*, as described in (iv).(b), unsets the `HasFired` flag and gives the token again to the Main process, which can only pass the token on to its children Wait and Send, after storing its current state in `nextState` and setting its state to backProp. Notice that, even if Main had enabled transitions, it would not have been allowed to execute

Listing 2.10 DSTM encoding schema – part 1: global declarations

```
1: #define MAX_PROC 255; // maximum number of concurrent processes
2: // Global variables, channels, datatypes declarations
3: bit x; int y; chan c = [8] of bit; ...
4: // Mtype declarations for each machine's state name
5: mtype = {S1, S2, S3, ..., backProp};
6: // Data objects needed to properly model the system
7: bit isFirstDescent = 1;
8: bit HasToken[MAX_PROC];
9: bit HasFired = 0;
10: bit dyingPid[MAX_PROC];
11: bit HasExecuted[MAX_PROC]; //set if pid executed in current step
12: bit descendantExecuted[MAX_PROC];
13: bit updateState = 0;
14: //structure needed to keep track of the process hierarchy
15: typedef childrenArray {
16:     bit children[MAX_PROC];
17: }
18: childrenArray ChildrenMatrix[MAX_PROC];
```

them anyway, since its `DescendantExecuted` flag is set. Similarly, Send is not allowed to execute any transition despite the fact that it owns the token and has an enabled transition (T11), and is only allowed to consume its token, store its next state in `nextState` and set its state to backProp. When Wait is scheduled, it can perform transition T6, since it owns the token and its `DescendantExecuted` flag is unset. After executing the transition, the process' `DescendantExecuted` and the `HasFired` flags are set, the next state is stored in `nextState` and `state` is set to `backProp`. After that, Wait is allowed to execute and propagate to its parent Main the `DescendantExecuted` flag. As in the previous phase, after the deadlock, every process resets its state to its intended value and a new phase starts. During this phase no process can execute, and the system eventually reaches a deadlock state with the `HasFired` flag set to false. After restoring the correct states for the next step (see (iv).(c)), *Engine* concludes the current step – which now includes both transition T10 and T6 and is indeed maximal – and starts the next one (see (iv).(d)).

2.4.5 Mapping a DSTM model to a Promela specification

With a mechanism to correctly implement steps semantics in place, it is now possible to extend the mapping schema described in Subsection 2.4.3 to obtain Promela encodings of hierarchical DSTM models. Consider a DSTM model $D = \langle M_1, \ldots, M_n, X, C, P \rangle$ and suppose the flattening step has been applied to it, obtaining n flat machines. The corresponding Promela encoding for D is quite similar to the one described in 2.4.3 and is described as follows.

The first part of the Promela specification contains declarations of global variables, channels and datatypes. This includes both data objects used in the DSTM model and data objects required to properly model the system. Listing 2.10 shows the general schema for

this part of the specification. The schema is very similar to the one described in 2.4.3, with the main difference being the addition of the flags mentioned in the previous subsection and the newly-introduced data structure `ChildrenMatrix`, a square matrix of `bits` of size `MAX_PROC`, declared in lines 15–18. This structure is needed to keep track of the process hierarchy and is used in such a way that the `bit ChildrenMatrix[A].children[B]` is set iff the process with pid `B` is a child of the process with pid `A`.

The second part of the specification contains *n* `proctypes`, each one obtained as described in the schema shown in Listing 2.11. The generic `M` proctype has the same parameters as the corresponding flattened machine, and starts with declaration of local variables and channels required for termination synchronization with its children, if any. Then, the process enters the main repetition construct (line 6), exiting only when a termination signal is received on the dedicated channel `chTerm` or when a its parent pid is marked as "dying" (see the unless construct at line 40). Inside the main repetition construct there is one option sequence for each machine state `S`, having guard `(state == S && HasToken[_pid]==1)`. Each option sequence immediately consumes the token (line 9) and then enters a selection construct to (possibly non-deterministically) choose a transition to be executed. This transition selection construct contains an option sequence for each transition *t* exiting the state `S`. Each option sequence is guarded by a condition having the form `(`ξ `&&` ϕ `&& !DescendantExecuted)`, with ξ and ϕ being respectively the trigger and the guard associated with the transition, and performs the required actions specified in the transition's decoration. For each *run* statement of the form `run P(X,init,chTerm,chTerm_ex1,...)` (where `X` is either `_pid` or `parent`, in case of asynchronous forks) included in the actions, the pid of the newly-instantiated process is stored in a temporary variable (`pidTemp = run P(...)`) and a statement to keep track of the process hierarchy is added (`ChildrenMatrix[X].children[pidTemp]=1`). Moreover, each option sequence takes care of setting the `state` variable to the corresponding transition's target and setting the flags `HasFired`, `HasExecuted`, and `DescendantExecuted` to *true* (lines 32–33). An additional option sequence, enabled only when none of the transitions is enabled, takes care of passing the token to the process' children. This is necessary only if the process has not executed directly during the current step. After the selection construct, the value in the `state` variable is stored in the `nextState` variable and `state` is set to `backProp`. Before ending the main repetition construct, two other option sequences are added to model `DescendantExecuted`'s backpropagation and the restoration of the intended `state` value before the next phase or step. The first one (line 52-53) is executable if `state == backProp`, `DescendantExecuted` is set, and `didBackProp` is unset. The `didBackProp` flag is used to make sure that this option sequence is fired at most once per phase. The second one (line 55-56) is needed to restore `state` to a meaningful value before the next phase or step.

Listing 2.12 shows a Promela encoding for the *Engine* process and concludes the specification. `Engine`, after declaring local variables and channels required for termination synchronization, starts an instance of the `Main` process (line 5) and registers it as its child (line 6). Notice that the `Main` process won't be able to perform any transition until it has received a token. `Engine` then starts a new step by executing the sequence labelled

Listing 2.11 DSTM encoding schema – part 2: flat machine to proctype

```
1: proctype M(pid parent;mtype initial;chan chTerm;chan chTerm_ex) {
2:   bit didBackProp = 0; byte i;
3:   //declare channels for termination synch. with children here
4:   mtype state=initial, nextState;
5:
6:   do
7:   // for each state S ∈ N_i ∪ En_i
8:   :: (state == S && HasToken[_pid]) -> atomic {
9:     HasToken[_pid]=0;
10:    didBackProp=0;
11:    if
12:    // for each trans. t with Src_1(t) = S, Trg_1(t) = T, Dec_1(t) = ⟨ξ, φ, α⟩
13:    :: (ξ && φ && !descendantExecuted[_pid]) ->
14:        α; state = T; HasFired=1;
15:        HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
16:    // other transitions here
17:    :: else -> //no transition is executable
18:      if
19:      :: (!HasExecuted[_pid]) ->
20:        // if this proc did not exec. in this step
21:        for (i : 0 .. MAX_PROC-1) { // pass token to children
22:          if
23:          ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
24:          ::else->skip;
25:          fi;
26:        }
27:      ::else->skip;
28:      fi;
29:    fi;
30:    nextState = state; state = backProp;
31:  }
32:  // other states here
33:  // handle upwards propagation of descendantExecuted
34:  ::(state==backProp && descendantExecuted[_pid] && !didBackProp)
35:    -> { didBackProp = 1; descendantExecuted[parent] = 1 }
36:  //handle original state restoring after backProp
37:  ::(state==backProp && updateState) ->
38:    { state = nextState; didBackProp=0 }
39:
40:  od unless (chTerm?[1] ||  dyingPid[parent]) -> {
41:    chTerm?1; dyingPid[_pid]=1
42:  }
43: }
```

Listing 2.12 DSTM encoding schema – part 3: the Engine proctype

```
1: active proctype Engine() {
2:   pid PidMain; byte i;
3:   chan chTerm_Main = [1] of {bit};
4:   chan chTerm_Main_exit = [1] of {bit};
5:   PidMain = run Main(_pid,initial,chTerm_Main,chTerm_Main_exit);
6:   ChildrenMatrix[_pid].children[PidMain]=1;
7:
8:   nextStep: // starts a new step
9:     atomic {
10:      // handle external channels management
11:      updateState=0
12:      HasFired=0;
13:      isFirstDescent=1;
14:      for (i : 0 .. MAX_PROC-1){
15:        HasExecuted[i]=0;
16:        descendantExecuted[i]=0;
17:        HasToken[i] = ChildrenMatrix[_pid].children[i];
18:      }
19:    }
20:    goto waitTimeout;
21:
22:  nextPhase: // starts a new phase in the current step
23:    atomic {
24:      updateState=0;
25:      HasFired=0;
26:      for ( i : 0 .. MAX_PROC - 1){
27:        // give token to engine's children
28:        HasToken[i] = ChildrenMatrix[_pid].children[i];
29:      }
30:      isFirstDescent = 0; //It's at least the second one
31:    }
32:    goto waitTimeout;
33:
34:  waitTimeout:
35:    do
36:    :: timeout -> //deadlock
37:      if
38:      :: (!HasFired && isFirstDescent) -> goto abort;
39:      :: (!HasFired && !isFirstDescent && !updateState) ->
40:        updateState = 1;
41:      :: (!HasFired && !isFirstDescent && updateState) ->
42:        goto nextStep;
43:      :: (HasFired && !updateState) -> updateState = 1;
44:      :: (HasFired && updateState) -> goto nextPhase;
45:      fi;
46:    od;
47:
48:  abort:
49:    dyingPid[_pid]=1;
50: }
```

Listing 2.13 Coverage-requiring never claim

```
1: never {
2:     never_step:
3:         if
4:         :: (LastState==S) -> goto end_never
5:         // resp. (LastTransition==t) -> goto end_never
6:         :: else -> goto never_step
7:         fi;
8:     end_never: skip
9: }
```

`nextStep` (lines 8–20). This sequence performs management operations on external channels, properly initializes the flags and then passes the token on to the `Main` process and to its siblings, if any (line 17). After that, with **`goto`** `waitTimeout` at line 20, `Engine` starts the `waitTimeout` sequence, where it blocks until the system reaches a deadlock state. Once a deadlock occurs, the selection construct in lines 37–45 models the choices already detailed in point (iv) at page 51. The `nextPhase` sequence, as its name suggests, takes care starting a new phase by reinitializing flags and passing the token again to the main process and its siblings. The `abort` sequence just marks `Engine`'s as dying and this has a ripple effect causing the termination of all processes (see line 40 in Listing 2.11). A complete example of Promela mapping for the *counting* DSTM model detailed in Figure 1.1 and in Table 1.1 is given in Appendix A.

2.5 Test case generation

In this section a procedure to generate test cases covering specific states or transitions from the Promela encoding of a DSTM model is discussed. The main intuition is to translate a state (or transition) coverage request to a `never` claim, and then use Spin to check if there exists a run satisfying the coverage request. If such run exists, the Spin-generated trail file can be inspected in simulation mode to generate the required test case, otherwise the state/transition is proved unreachable.

A coverage request can be translated to a `never` claim quite simply as follows. Two additional global, `mtype` typed, variables `LastState`, `LastTransition` are introduced in the Promela specification, along with the necessary `mtype` declarations for transition names. Those variables are then updated every time a transition fires so that they always hold respectively the value corresponding to the latest entered state and to the latest fired transition. After these changes, the `never` claim corresponding to a certain state or transition coverage request is as simple as the one shown in Listing 2.13.

–3–
Reasoning about Hierarchical Concurrent Computations with Interrupts

CONTENTS: **3.1 Hierarchical Temporal Logic with Interrupts. 3.2 Communicating Structured Automata with Interrupts. 3.3 Deciding CHA$^{\lightning}$ emptiness. 3.4 Satisfiability of HLTL$^{\lightning}_{\mathrm{L}}$ over hierarchical computations. 3.5 HLTL$^{\lightning}_{\mathrm{L}}$ Model Checking.**

The DSTM formalism, as discussed throughout Chapter 1, models hierarchical concurrent systems whose global state is described by tree-like structures like the one in Definition 9. Unlike other formalisms such as Communicating Structured Systems, originally introduced in [22], DSTM is also capable of modelling interrupts, i.e. the abrupt termination of computation subtrees of arbitrary height when an interrupting event occurs.

This chapter presents, in Section 3.1, Hierarchical Linear-time Temporal Logic with Interrupts (HLTL$^{\lightning}$), an extension of the well-known Linear-time Temporal Logic (LTL) [23] designed to express linear properties of hierarchical interrupting systems. A concrete instantiation HLTL$^{\lightning}$ semantics on Communicating Structured Automata with Interrupts (CSA$^{\lightning}$) is then given in Section 3.2. After that, Section 3.3 shows a decision procedure for the emptiness problem on CSA$^{\lightning}$, and Section 3.4 provides an automata-based solution to the satisfiability problem of the local fragment of HLTL$^{\lightning}$. Finally, in Section 3.5, a model checking procedure for CSA$^{\lightning}$ models against HLTL$^{\lightning}$ specification is provided.

3.1 Hierarchical Temporal Logic with Interrupts

In [24], Benerecetti et al. introduced Hierachical LTL (HLTL), a novel extension of the classic Linear-time Temporal Logic (LTL) designed to express linear properties of hierarchical systems. As known, LTL allows for reasoning about infinite sequences of

unstructured (flat) states. HLTL, on the other hand, allows for reasoning about sequences of tree-structured states and is able to explicitly reference the tree-like structure of each state. The main intuition behind HLTL is to use classic LTL operators to reason about the evolution of a given module, while additional operators are used to contextualize formulae in the hierarchy of activated modules. A HLTL formula is locally evaluated with regard to a context (i.e. a given module, corresponding to a vertex in the tree-like hierarchical structure of the current state), and the context can change during the evaluation of a formula by moving along both the vertical and the horizontal dimension. The vertical dimension is related to the hierarchy (caller/called relations), while the horizontal one is related to concurrency (left/right sibling in the tree).

Suppose that the current context is a vertex t, corresponding to the state of a given module committed to a call, which is to say that t has children (the modules it invoked) in the current state. It is possible to express the fact that the formula ϕ is required to hold in the i-th child of t by means of the formula $\downarrow_i(\phi)$. Notice that in HLTL it is possible to navigate the vertical dimension only downwards. If the context t has siblings in the current state (i.e. is executing concurrently with other modules as a result of a call operation), then it is possible to express the fact that the formula ϕ is required to hold in its left (resp. right) sibling with the formula $\leftarrow(\phi)$ (resp. $\rightarrow(\phi)$).

These vertical and horizontal displacement operators can be freely combined with linear temporal operators that allow for expressing behavioural properties of a module with regard to the temporal dimension. HLTL defines two different versions of each linear temporal operator: a *local* version with subscript l and a *global* one with subscript g. The local next $\mathtt{X}_l$, for example, captures a local notion of successor, in which the context is directly involved in the system's evolution. The global next $\mathtt{X}_g$, on the other hand, corresponds to a notion of successor sensitive to any evolution affecting the subtree rooted in the current context. These different interpretations of next are formalized in Definition 13.

In this section, HLTL itself is extended with an additional *interrupting next* operator, in symbols $\mathtt{X}_\lightning$, which can be used to explicitly predicate about interrupts. Considering the context t described above, the formula $\mathtt{X}_\lightning(\phi)$ holds in t iff the module corresponding to t locally performs an interrupting transition resulting in a state in which ϕ holds. The resulting logic is called Hierarchical Temporal Logic with Interrupts (HLTL$^\lightning$).

Definition 11 (HLTL$^\lightning$ syntax)**.** HLTL$^\lightning$ formulae are inductively defined as follows:

$$\phi ::= \top \mid p \in \mathcal{P} \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \downarrow_n(\phi) \mid \leftarrow(\phi) \mid \rightarrow(\phi) \mid \mathtt{X}_x\,\phi \mid \mathtt{X}_\lightning\,\phi \mid \phi\,\mathtt{U}_x\,\phi \mid \phi\,\mathtt{R}_x\,\phi,$$

where $\mathcal{P}$ is a set of atomic propositions and $x \in \{l, g\}$. HLTL$^\lightning_{\mathrm{L}}$ denotes the local fragment, having only operators with $x = l$.

A HLTL$^\lightning$ formula is interpreted over interrupting hierarchical words, which are sequences of tuples consisting in a labelled trees, a *frontier* and a set of *interrupting vertices*. An interrupting hierarchical word can be seen as a computation of a hierarchical modular system with interrupts, with each symbol's labelled tree representing the current

state of the modular system. A *frontier* is a subset of the vertices in the labelled tree and represents those modules which are synchronously participating in the computation step whose target is the next symbol in the hierarchical word. The set of *interrupting nodes* is a subset of the frontier and includes only those modules which are participating in the current computation step by performing interrupting transitions. If a module in the tree descends from a module belonging to the frontier, it will be deallocated in the next system state (because it either has terminated or is interrupted by its ancestor in the frontier). If a module does not belong to the frontier and is not a descendant of a node belonging to the frontier, then it is not affected at all by the current computation step.

Before formalizing these concepts it is necessary to introduce some new notation. Much like DSTM control trees in Section 1.3.3, a tree $T \subseteq \mathbb{N}^*$ is defined as a prefix-closed set of finite sequences of natural numbers, with the empty sequence being denoted by ε. Furthermore, for any set of vertices $S \subseteq T$, the set of children of S is defined as $chl(S) = \{t \cdot i \in T \mid t \in S \wedge i \in \mathbb{N}\}$. For a singleton set $\{t\}$, the abbreviated notation $chl(t)$ shall be used. The set of leaves of T is defined as $lvs(T) = \{t' \in T \mid chl(t) = \varnothing\}$, i.e. the set of those vertices having no child. The set of descendants of S is defined as $des(S) = \{t \cdot t' \in T \mid t \in S \wedge t' \in \mathbb{N}^*\}$.

Definition 12 (Interrupting hierarchical word)**.** An interrupting hierarchical word over the alphabet Σ is a sequence of the form

$$\langle (T_0, v_0), Fr_0, In_0 \rangle, \langle (T_1, v_1), Fr_1, In_1 \rangle, \ldots, \langle (T_i, v_i), Fr_i, In_i \rangle, \ldots$$

such that, for all $i \geq 0$:

(i) (T_i, v_i) is a labelled tree over Σ, with $v_i : T_i \mapsto \Sigma$;

(ii) the frontier $Fr_i \subseteq T_i$, and $Fr_i \cup (T_i \setminus Des(Fr_i)) \subseteq T_{i+1}$;

(iii) for all $t \in (T_i \setminus Des(Fr_i))$, $v_i(t) = v_{i+1}(t)$;

(iv) the interrupting vertices set $In_i \subseteq Fr_i$ and $In_i \cap lvs(T_i) = \varnothing$.

In the above definition, (ii) requires that, for each symbol in the interrupting hierarchical word, the frontier is a subset of vertices of the labelled tree and that all vertices in the labelled tree which either belong to the frontier or do not have a node belonging to the frontier as a proper prefix, are also vertices of the next symbol's tree. Moreover, (iii) requires that the labelling for the vertices which do not belong to the frontier and do not descend from nodes belonging to it, remains unchanged, accordingly with the fact that in a concurrent hierarchical system modules not performing any action are expected not to change their state. Point (iv) guarantees the well-formedness of the *interrupting vertices* set, i.e. that the set of vertices participating in the current step by performing interrupting transitions is included in the set of vertices participating in the current computation step and consists solely of vertices having at least one child, as child-less vertices cannot perform interrupting transition since they have no child module to interrupt.

When dealing with sequences of states, the standard interpretation of linear temporal operators has a global character, i.e. the concept of *next* state is relative to the overall system state. Given a context $c = \langle (T_i, v_i), t \rangle$, with $t \in T_i$, the standard next from c would single out as the successor the context $\langle (T_{i+1}, v_{i+1}), t \rangle$, if $t \in T_{i+1}$. With flat, unstructured states, this classic interpretation is perfectly adequate, but it falls short when dealing with sequences of structured states of concurrent hierarchical modular systems, as it is not able to fully capture the concurrent nature of computations. In a concurrent hierarchical computation, the step leading from $\langle (T_i, v_i), t \rangle$ to $\langle (T_{i+1}, v_{i+1}), t \rangle$ could not involve at all t or its descendants. To better capture the concurrent and hierarchical nature of computations, as already anticipated, two additional interpretations of linear temporal operators are introduced.

Definition 13 (Interpretations of *next*)**.** Given an interrupting hierarchical word

$$\xi = \xi_0, \xi_1, \ldots, \xi_k, \ldots,$$

with $\xi_i = \langle (T_i, v_i), Fr_i, In_i \rangle$, for all $i \geq 0$ and $t \in T_i$:

- the *global* $Next_g(\xi_i, t)$ is the hierarchical computation symbol ξ_j (if any) such that $j > i$ is the least index such that $t' \in Fr_{j-1}$, for some $t' \in T_{j-1} \cap Des(t)$ and, for each $i < \ell < j$ there is no prefix t' of t with $t' \in Fr_\ell$.
- the *local* $Next_l(\xi_i, t)$ is the hierarchical computation symbol ξ_j (if any) such that $j > i$ is the least index such that $t \in Fr_{j-1}$ and, for each $i < \ell < j$ there is no prefix t' of t with $t' \in Fr_\ell$.

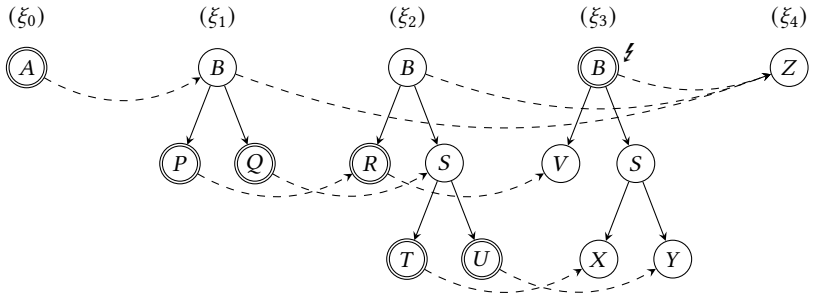
As usual, $Next_x^*(\xi_i, t)$, with $x \in \{l, g\}$, denotes the reflexive and transitive closure of $Next_x^*(\xi_i, t)$ and is defined inductively as the set of hierarchical symbols such that:

(i) $\xi_i \in Next_x^*(\xi_i, t)$;

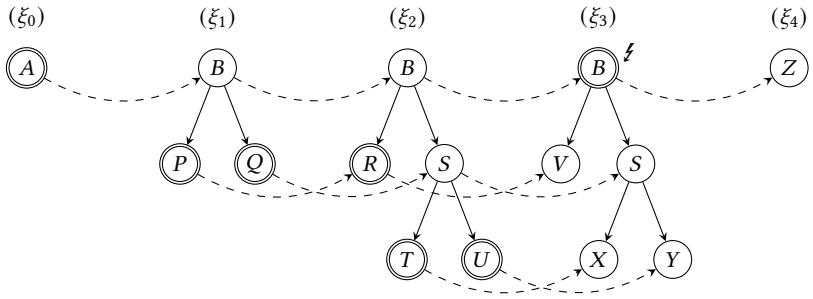
(ii) $\xi_j \in Next_x^*(\xi_i, t)$ iff $\xi_j \in Next_x(\xi_k, t)$ and $\xi_k \in Next_x^*(\xi_i, t)$.

Note that, when the context is fixed in the root of the hierarchy, the *global* interpretation of next is in fact the same as the classic interpretation.

Example 12 (An interrupting hierarchical word and the interpretations of *next*)**.** Consider the interrupting hierarchical word $\xi = \xi_0, \xi_1 \ldots$, whose first five symbols are represented in Figure 3.1. In the proposed depiction, each tree node is decorated with its labelling and is drawn with a double circle if it belongs to the frontier. Vertices belonging to the *interrupting set* are furthermore decorated with a lightning symbol. First of all, it is trivial to check that ξ is indeed a well-formed interrupting hierarchical word as it satisfies all the constraints in Definition 12. In Subfigures 3.1a and 3.1b, the dashed oriented edges exiting tree vertices represent, respectively, the $Next_l$ and the $Next_g$ relations. In node ε in ξ_0, both the local and the global next map to ξ_1's root. In ξ_1, nodes 1 and 2 belong to the frontier and thus their local and global next single out nodes 1 and 2 in ξ_2. Node ε



(a) The $Next_l$ relation



(b) The $Next_g$ relation

Figure 3.1: An interrupting hierarchical word decorated with both $Next_l$ and $Next_g$

is not participating directly in the computation step leading from ξ_1 to ξ_2, and its local next maps to symbol ξ_4, accordingly to Definition 13. Since its descendants 1 and 2 are participating in the current step, ε's global next maps to symbol ξ_2. In ξ_2, node 2 has no local next but has a global next in symbol ξ_3, as its descendants are participating in the current computation step. As in ξ_2, node ε does not participate directly to the current computation step and its local and global next map, respectively, to symbols ξ_4 and ξ_3. In the last computation step, node ε belongs to the set of *interrupting* modules and performs an interrupting transition deallocating all of its descendants. Both its local and global next map to symbol ξ_4.

With the definition of interrupting hierarchical word and the interpretations of *next* in place, it is now possible to formalize HLTL$^{\lightning}$ semantics.

Definition 14 (HLTL-i semantics)**.** The satisfaction of an HLTL-i formula ϕ in node $t \in T_i$ at the i-th symbol of an interrupting hierarchical word $\xi = \xi_0, \xi_1, \ldots, \xi_k, \ldots$, with $\xi_i = \langle (T_i, v_i), Fr_i, In_i \rangle$, is defined recursively as follows:

- $\langle \xi_i, t \rangle \vDash p$ iff $p \in v_i(t)$, with $p \in \mathcal{P}$;
- Boolean connectives are defined as usual;
- $\langle \xi_i, t \rangle \vDash \downarrow_j \phi$ iff $t \cdot j \in T_i$ and $\langle \xi_i, t \cdot j \rangle \vDash \phi$;

- $\langle \xi_i, t \rangle \vDash \leftarrow\!\!(\phi)$ iff $t = t' \cdot j$, with $j > 1$, and $\langle \xi_i, t' \cdot (j-1) \rangle \vDash \phi$;
- $\langle \xi_i, t \rangle \vDash \rightarrow\!\!(\phi)$ iff $t = t' \cdot j$, $t' \cdot (j+1) \in T_i$, and $\langle \xi_i, t' \cdot (j+1) \rangle \vDash \phi$;
- $\langle \xi_i, t \rangle \vDash \mathtt{X}_x(\phi)$, with $x \in \{l, g\}$, iff there exists ξ_j such that $\xi_j = \mathit{Next}_x(\xi_i, t)$ and $\langle \xi_j, t \rangle \vDash \phi$.
- $\langle \xi_i, t \rangle \vDash \mathtt{X}_\ell(\phi)$ iff there exists ξ_j such that $\xi_j = \mathit{Next}_l(\xi_i, t)$, $\langle \xi_j, t \rangle \vDash \phi$ and $t \in \mathit{In}_{j-1}$;
- $\langle \xi_i, t \rangle \vDash \phi\, \mathtt{U}_x\, \psi$, with $x \in \{l, g\}$, iff there exists $\xi_j \in \mathit{Next}_x^*$ such that $\langle \xi_j, t \rangle \vDash \psi$ and, for all $\xi_k \in \mathit{Next}_x^*$, with $i \leq k < j$, $\langle \xi_k, t \rangle \vDash \phi$.
- $\langle \xi_i, t \rangle \vDash \phi\, \mathtt{R}_x\, \psi$, with $x \in \{l, g\}$, iff for all $\xi_j \in \mathit{Next}_x^*(\xi_i, t)$, if $\langle \xi_j, t \rangle \nvDash \psi$, then there exists $\xi_k \in \mathit{Next}_x^*$, with $i \leq k < j$, such that $\langle \xi_k, t \rangle \vDash \phi$.

A hierarchical word ξ satisfies a HLTL$^\ell$ formula ϕ, in symbols $\xi \vDash \phi$, if ϕ holds in the word's first symbol's root, i.e. $\langle \xi_0, \varepsilon \rangle \vDash \phi$.

The following abbreviations will be used hereafter: $\bot$ for $\neg\top$; $\phi \Rightarrow \psi$ for $\neg\phi \vee \psi$; Stop_x for $\neg\, \mathtt{X}_x\, \top$, with $x \in \{l, g\}$, expressing the fact that the current context has no local or global future. Derived temporal operators *eventually* $\mathtt{F}_x$ and *globally* $\mathtt{G}_x$, with $x \in \{l, g\}$, can be defined as follows. $\mathtt{F}_x\, \phi$, requiring that in some future point in Next_x^* ϕ holds, is equivalent to $\top\, \mathtt{U}_x\, \phi$. $\mathtt{G}_x\, \phi$, requiring for ϕ to hold in all points in Next_x^*, is equivalent to $\bot\, \mathtt{R}_x\, \phi$.

Furthermore, the "interrupting" versions of all temporal operators, which consider only interrupting transitions, can be defined by combining the interrupting next operator with standard temporal operators. The interrupting until operator $\phi\, \mathtt{U}_\ell\, \psi$ can be expressed as $[\phi \wedge \mathtt{X}_\ell\, (\phi \vee \psi)]\, \mathtt{U}_l\, \psi$. Similarly, the interrupting release operator $\phi\, \mathtt{R}_\ell\, \psi$ is equivalent to $\phi\, \mathtt{R}_l\, [\psi \wedge ((\mathtt{X}_\ell\, \top) \vee \phi)]$. The interrupting globally operator $\mathtt{G}_\ell\, \phi$ is equivalent to $\phi \wedge \mathtt{G}\, (\mathtt{X}_\ell\, \phi)$, and the interrupting eventually operator $\mathtt{F}_\ell\, \phi$ as $(\mathtt{X}_\ell\, \top)\, \mathtt{U}\, \phi$.

Example 13 (HLTL$^\ell$ formulae)**.** Consider the interrupting hierarchical word ξ depicted in Figure 3.1 and discussed in Example 12. It holds that $\langle \xi_{i \in \{1,2,3\}}, \varepsilon \rangle \vDash \mathtt{X}_\ell\, Z$, since in each of these contexts, the local next $\langle \xi_4, \varepsilon \rangle \vDash Z$ and ε belongs to the set of interrupting vertices in ξ_3. The HLTL$^\ell$ formula $\phi = (A \vee B)\, \mathtt{U}_g(\mathtt{X}_\ell\, Z)$ is satisfied by ξ, i.e. $\langle \xi_0, \varepsilon \rangle \vDash \phi$, since there exists the symbol ξ_1 in $\mathit{Next}_g^*(\xi_0, \varepsilon)$ such that $\langle \xi_3, \varepsilon \rangle \vDash \mathtt{X}_\ell\, Z$ and for $i \in \{0, 1, 2\}$, $\langle \xi_i, \varepsilon \rangle \vDash (A \vee B)$. The HLTL$^\ell$ formula $\psi = \mathtt{G}_g\, \big((\downarrow_1(R)) \Rightarrow \big(\downarrow_2(\mathit{Stop}_l)\big)\big)$, requires that, in each state of the computation, if the first child of the primary module satisfies R, then the second child has no local next. It holds that $\xi \vDash \psi$. The HLTL$^\ell$ formula $\mu = \mathtt{X}_l\, (\downarrow_1 (P \Rightarrow \rightarrow (\mathtt{X}_l(S))))$, requiring that in the local next relative to its context, if the first child satisfies P, then its right sibling has a local next satisfying S, is also satisfied in ξ.

Definition 15 (Negation-normal form (NNF) for HLTL formulae)**.** An HLTL formula ϕ is in negation-normal form if the negation operator $\neg$ is only applied to atomic propositions and subformulae in $\{\rightarrow \top, \leftarrow \top, \mathtt{X}\, \top, \mathtt{X}_\ell\, \top\}$.

Any HLTL formula can be transformed into an equivalent formula in NNF by "pushing" the negations inwards. This is done by applying the following equivalences from left to right as long as possible

$$\begin{aligned}
\neg\neg\phi &\equiv \phi \\
\neg(\phi \vee \psi) &\equiv \neg\psi \wedge \neg\psi \\
\neg(\phi \wedge \psi) &\equiv \neg\psi \vee \neg\psi \\
\neg(\phi \,\mathtt{U}_x\, \psi) &\equiv \neg\psi \,\mathtt{R}_x\, \neg\psi \\
\neg(\phi \,\mathtt{R}_x\, \psi) &\equiv \neg\psi \,\mathtt{U}_x\, \neg\psi \\
\neg(\mathtt{Op}(\varphi)) &\equiv \mathtt{Op}(\neg\varphi) \vee \neg\mathtt{Op}\top
\end{aligned}$$

with $x \in \{l, g\}$, $\mathtt{Op} \in \{\leftarrow, \rightarrow, \mathtt{X}, \mathtt{X}_{\not\iota}, \downarrow_{i\in\mathbb{N}^+}\}$.

3.2 Communicating Structured Automata with Interrupts

HLTL$^{\not\iota}$ allows to express linear properties of hierarchical words, which – as previously discussed – can be seen as computations of possibly-interrupting concurrent hierarchical systems, including DSTM specifications. This section provides a possible instantiation of the concrete semantics of HLTL$^{\not\iota}$ in terms of Communicating Structured Automata with Interrupts (CSA$^{\not\iota}$), by extending the concrete semantics for HLTL given by Benerecetti et al. in [24] in terms of Communicating Structured Automata. CSA$^{\not\iota}$ are simpler systems than DSTM, yet maintain the most important characteristics of hierarchy, concurrency and the possibility of interrupts, and therefore are better-suited for a preliminary study like the one in this chapter.

Similarly to DSTM, control states in CSA$^{\not\iota}$ are partitioned into locations and boxes, depending on whether the control state is refined by other machines. Locations, much like DSTM, can be qualified as *entering* or *exiting* and each CSA$^{\not\iota}$ has at least one entering location. A transition entering a box is interpreted as a procedure call activating a sequence of CSA$^{\not\iota}$, possibly containing multiple instances of the same automaton, which after activation run synchronously in parallel. Such sequence of automata to be activated is associated with the box by means of a refinement function β. A transition entering a box b specifies, for each of the automata in $\beta(b)$, an entering state, in symbols $(b, en_1, \ldots, en_k)$. A synchronous return from a box call specifies, for each of the automata associated with box, the corresponding exiting state $(b, ex_1, \ldots, ex_k)$. A return by interrupt, on the contrary, does not specify a sequence of exiting states. In other words, the source of a transition in a CSA$^{\not\iota}$ is either a location or a box (in case of return by interrupt) or a tuple of the form $(b, ex_1, \ldots, ex_k)$ (in case of synchronous return), while the target of a CSA$^{\not\iota}$ transition is either a location or a tuple of the form $(b, en_1, \ldots, en_k)$. Parallel components refining a common box synchronize on communication symbols from a set Γ, i.e. if one

of the components performs a transition labelled with the synchronization symbol $\gamma \in \Gamma$, then all other components must be able to perform a transition labelled with γ. Active components called by different boxes, on the other hand, need not to communicate and evolve accordingly to an interleaving semantics.

More formally, given a set Σ of input symbols, a set Γ of synchronization symbols, a set of states Q and a set of boxes $B \subseteq Q$, the class $\text{CSA}^{\lightning}(\Sigma, \Gamma, Q, B)$ of Communicating Structured Automata with Interrupts over Σ, Γ, Q, B is defined as follows.

Definition 16 (Communicating Structured Automata with Interrupts ($\text{CSA}^{\lightning}$) [24])**.** Let $S = Q \cup (B \times Q^+)$ and $L \triangleq Q \setminus B$ be, respectively, the sets of structured states and locations. The class of Concurrent Structured Automata with Interrupts over Σ, Γ, Q, B, is the maximal set

$$\text{CSA}^{\lightning}(\Sigma, \Gamma, Q, B) \subseteq 2^Q \times 2^L \times 2^L \times 2^B \times (B \rightharpoonup CSA(\Sigma, \Gamma, Q, B)^+) \times (S \times \Sigma \times \Gamma \rightharpoonup 2^S) \times 2^Q \times 2^{2^Q}$$

such that each tuple $\mathscr{A} = \langle Q_{\mathscr{A}}, En_{\mathscr{A}}, Ex_{\mathscr{A}}, B_{\mathscr{A}}, \beta_{\mathscr{A}}, \delta_{\mathscr{A}}, F_{\mathscr{A}}, R_{\mathscr{A}} \rangle \in \text{CSA}^{\lightning}(\Sigma, \Gamma, Q, B)$ satisfies the following:

(i) the sets of entering and exiting locations are disjoint subsets of the set of locations belonging to $Q_{\mathscr{A}}$, in symbols $En_{\mathscr{A}} \cup Ex_{\mathscr{A}} \subseteq Q_{\mathscr{A}} \cap L$ and $En_{\mathscr{A}} \cap Ex\,\mathscr{A} = \emptyset$;

(ii) the box refinement function $\beta_{\mathscr{A}}$ is such that its domain is $B_{\mathscr{A}}$;

(iii) the non-deterministic transition function $\delta_{\mathscr{A}}$ is a partial function $Src_{\mathscr{A}} \times \Sigma \times \Gamma \rightharpoonup 2^{Trg_{\mathscr{A}}}$, where:

$$Src_{\mathscr{A}} \triangleq (L \setminus Ex_{\mathscr{A}}) \cup \left(B_{\mathscr{A}} \times_b \prod_{i=1}^{|\beta(b)|} Ex_{\beta_{\mathscr{A}}(b)_i} \right) \cup B_{\mathscr{A}}$$

$$Trg_{\mathscr{A}} \triangleq (L \setminus En_{\mathscr{A}}) \cup \left(B_{\mathscr{A}} \times_b \prod_{i=1}^{|\beta(b)|} En_{\beta_{\mathscr{A}}(b)_i} \right);$$

(iv) The acceptance conditions $F_{\mathscr{A}} \subseteq Q_{\mathscr{A}}$ and $R_{\mathscr{A}} \subseteq 2^{Q_{\mathscr{A}}}$.

The semantics of acceptance conditions is formally defined in Definition 23.

Example 14 (A simple $\text{CSA}^{\lightning}$)**.** Consider the $\text{CSA}^{\lightning}$ $\mathscr{A}$ depicted in Figure 3.2. The proposed graphical formalism is very similar to the one used with DSTM: entering nodes are depicted as circles, locations as rounded rectangles, boxes as rectangles and exiting locations as crossed-out circles. Structured states are depicted as entering (resp. exiting) nodes placed on the corresponding box's border and suitably labelled with a sequence of entering (resp. exiting) nodes of the automata instantiated by the box. Moreover, each state is labelled with its name, and each transition, represented with an oriented edge connecting the source and target (structured) states, is decorated with a semicolon-separated couple consisting of an input symbol and a synchronization symbol. Interrupting transitions are decorated with a

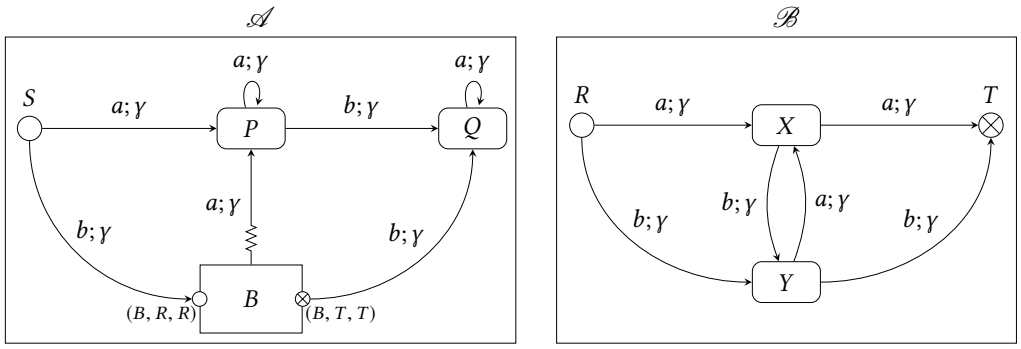


Figure 3.2: A simple CHA$^{\lightning}$

brief zig-zag segment. Formally, $\mathscr{A} = \langle Q_{\mathscr{A}}, En_{\mathscr{A}}, Ex_{\mathscr{A}}, B_{\mathscr{A}}, \beta_{\mathscr{A}}, \delta_{\mathscr{A}}, F_{\mathscr{A}}, R_{\mathscr{A}} \rangle$, with: the set of states $Q_{\mathscr{A}} = \{S, P, Q, B\}$; the set of entering locations $En_{\mathscr{A}} = \{S\}$; the set of exiting locations $Ex_{\mathscr{A}} = \emptyset$, which is not surprising since $\mathscr{A}$ is a top-level automaton; the set of boxes $B_{\mathscr{A}} = \{B\}$; the refinement function $\beta_{\mathscr{A}} = \{(B, \langle \mathscr{B}, \mathscr{B} \rangle)\}$, i.e. B instantiates two instances of the $\mathscr{B}$ automaton; $\delta_{\mathscr{A}} = \{(\langle S, a, \gamma \rangle, P), (\langle S, b, \gamma \rangle, (B, \langle R, R \rangle)), (\langle B, a, \gamma \rangle, P), (\langle \langle B, T, T \rangle, a, \gamma \rangle, Q), (\langle P, a, \gamma \rangle, P), (\langle P, b, \gamma \rangle, Q), (\langle Q, b, \gamma \rangle, Q)\}$; $F_{\mathscr{A}} = \{B\}$; $R_{\mathscr{A}} = \{\{Q\}\}$. In $\mathscr{A}$, the transition leading from the box B to the location P is an interrupting transition. The box B is refined by two instances of the $\mathscr{B}$ automaton, which is formalized as follows. $\mathscr{B} = \langle Q_{\mathscr{B}}, En_{\mathscr{B}}, Ex_{\mathscr{B}}, B_{\mathscr{B}}, \beta_{\mathscr{B}}, \delta_{\mathscr{B}}, F_{\mathscr{B}}, R_{\mathscr{B}} \rangle$, with: $Q_{\mathscr{B}} = \{R, X, Y, T\}$; $En_{\mathscr{B}} = \{R\}$; $Ex_{\mathscr{B}} = \{T\}$; $B_{\mathscr{B}} = \emptyset$; $\beta_{\mathscr{B}} = \emptyset$; $\delta_{\mathscr{B}} = \{(\langle R, a, \gamma \rangle, X), (\langle R, b, \gamma \rangle, Y), (\langle X, a, \gamma \rangle, T), (\langle X, b, \gamma \rangle, Y), (\langle Y, a, \gamma \rangle, X), (\langle Y, b, \gamma \rangle, T)\}$; $F_{\mathscr{B}} = \emptyset$; $R_{\mathscr{B}} = \{\{Y\}\}$. $\mathscr{A}$ and $\mathscr{B}$ belong to the class of CSA$^{\lightning}$ over the alphabet $\Sigma = \{a, b\}$, the synchronization symbols $\Gamma = \{\gamma\}$, the set of states $\{S, P, Q, B, R, X, Y, T\}$ and the set of boxes $\{B\}$.

Much like DSTMs (see Subsection 1.3.3), the complete configuration of a CSA$^{\lightning}$ in a given time instant is encoded by a tree-like structure, associating to each active module (vertex in the tree) its current control state and the corresponding automaton. Such structure, called *configuration tree*, is formalized in the following definition.

Definition 17 (Configuration tree [24])**.** Given a tree $T \subseteq \mathbb{N}^*$, a function $r : T \mapsto Q$ labelling each vertex of the tree with states, and a function $\mathfrak{m} : T \mapsto CSA(\Sigma, \Gamma, Q, B)$ associating to each point in T a machine, the tuple $\langle T, r, \mathfrak{m} \rangle$ is a configuration tree if the following conditions are satisfied:

(i) for all $t \cdot i \in T$ and $j < i$ with $i, j \in \mathbb{N}^+$, $t \cdot j \in T$;

(ii) for all $t \in T$, if $r(t) \in B_{\mathfrak{m}(t)}$ then $t \cdot k \in T$ with $k = |\beta_{\mathfrak{m}(t)}(b)|$ and $t \cdot k + 1 \notin T$;

(iii) for all $t \cdot i \in T$, $r(t) \in B_{\mathfrak{m}(t)}$ and $\mathfrak{m}(t \cdot i) = \left(\beta_{\mathfrak{m}(t)}(t)\right)_i$;

(iv) for all $t \in lvs(T)$, $r(t) \in Q_{\mathfrak{m}(t)} \cap L$.

In the above definition, constraint (i) requires for the tree to be well-formed, i.e. if a vertex has a k-th child, then it also has i-th children, for $i \in \{1, \ldots, k-1\}$. Constraints

(ii) and (iii) ensure that only box-labelled vertices have children, that each box has exactly as many children as the number of automata it instantiates, and that each i-th child corresponds to the i-th automata associated with the box by the refinement function. Finally, constraint (iv) requires that all leaves in the tree are labelled with locations, i.e. leaves cannot be labelled by boxes.

A computation step of a CSA$^{\ell}$ from one configuration tree to another involves a transition from all the nodes of a maximal set of children of some node t or from the root of the tree. This synchronous communication model among concurrently executing modules refining a common box is enforced by the definition of *frontier*. In what follows, after defining the *siblings relation* between elements of a configuration tree, the notion of CSA$^{\ell}$ frontier is formalized.

Given two vertices t_1, t_2 in a tree T, the sibling relation $\mathscr{S}$ holds between them ($t_1 \mathscr{S} t_2$) if there exists a $t \in T$ such that $t_1 = t \cdot i$ and $t_2 = t \cdot j$ for some $i, j \in \mathbb{N}$, $i \neq j$. The set of maximal sets of siblings in a tree T is defined as $MaxSib(T) = \{S \subseteq T \mid \forall t_1 \in S, t_2 \in T . t_2 \in S \Leftrightarrow t_1 \mathscr{S} t_2\}$.

Definition 18 (Frontier of a configuration tree)**.** Given a configuration tree $\langle T, r, \mathfrak{m} \rangle$, its frontier is defined as the collection of maximal sets of siblings such that, for each set of siblings, each component t is able to perform transitions, i.e. the predicate $canAct(t)$ holds. In symbols:

$$Front(\langle T, r, \mathfrak{m} \rangle) \triangleq \{S \in MaxSib(T) \mid \forall t \in S \,.\, canAct(t, \langle T, r, \mathfrak{m} \rangle)\} \,.$$

A component $t \in T$ is allowed to perform transitions if the following conditions hold:

(i) t is not in an exiting state of the corresponding machine, i.e. $r(t) \notin Ex_{\mathfrak{m}(t)}$;

(ii) t can either perform a synchronous transition or an interrupting one, which is to say that at least one of the following holds:

   (a) the children of t are all leaves and are labelled with exiting locations of the corresponding machines. Notice that this condition is also trivially satisfied by components whose state is a location, i.e. that have no children;

   (b) t is labelled with a box b of the corresponding machine and admits interrupting transitions, which is to say that there exists an input symbol $\sigma \in \Sigma$ and a synchronization symbol $\gamma \in \Gamma$ such that (b, σ, γ) belongs to the domain of the machine's transition function $dom(\delta_{\mathfrak{m}(t)})$. In this case, the predicate $interruptible(t, \langle T, r, \mathfrak{m} \rangle)$ holds.

In symbols, the *canAct* and the *interruptible* predicates are defined as follows.

$$canAct(t, \langle T, r, \mathfrak{m} \rangle) \triangleq r(t) \notin Ex_{\mathfrak{m}(t)} \wedge \left[\begin{array}{l} \big(chl(t) \subseteq lvs(T) \wedge \forall u \in chl(t).r(u) \in Ex_{\mathfrak{m}(t)}\big) \vee \\ interruptible(t) \end{array} \right]$$

$$interruptible(t, \langle T, r, \mathfrak{m} \rangle) \triangleq \big(r(t) \in B_{\mathfrak{m}(t)} \wedge (\{r(t)\} \times \Sigma \times \Gamma) \cap dom(\delta_{\mathfrak{m}(t)}) \neq \varnothing\big)$$

Intuitively, each non-empty set of siblings belonging to the frontier identifies a set of modules that can perform a transition in the automaton.

Definition 19 (CSA$^{\lightning}$ run)**.** A run (interrupting hierarchical computation) π of a CSA$^{\lightning}$ $\mathscr{A}$ over an interrupting hierarchical word $\xi = \langle (T_0, v_0), Fr_0, In_0\rangle, \langle (T_1, v_1), Fr_1, In_1\rangle, \ldots, \langle (T_i, v_i), Fr_i, In_i\rangle, \ldots$ is a sequence

$$\pi = \langle T_0, r_0, \mathfrak{m}_0\rangle \xrightarrow{\langle (T_0,v_0),Fr_0,In_0\rangle} \ldots \langle T_i, r_i, \mathfrak{m}_i\rangle \xrightarrow{\langle (T_i,v_i),Fr_i,In_i\rangle} \ldots$$

such that, for every $i \geq 0$:

(i) $Fr_i \in Front(\langle T_i, r_i, \mathfrak{m}_i\rangle)$;

(ii) $\forall t \in In_i$, $interruptible(t, \langle T_i, r_i, \mathfrak{m}_i\rangle)$ holds;

(iii) there exists $\gamma \in \Gamma$ such that:

  (a) for all $t \in In_i$, $r_{i+1} \in \delta_{\mathfrak{m}_i(t)}(r_i(t), v_i(t), \gamma)$;

  (b) for all $t \in Fr_i \setminus In_i$, if $r(t) \notin B_{\mathfrak{m}_i(t)}$ then $r_{i+1} \in \delta_{\mathfrak{m}_i(t)}(r_i(t), v_i(t), \gamma)$, else $r_{i+1} \in \delta_{\mathfrak{m}_i(t)}\left(\left(r_i(t), \prod_{t' \in chl(t)} r_i(t')\right), v_i(t), \gamma\right)$

(iv) for all $t \in T_i \setminus (Des(Fr_i))$, $r_{i+1}(t) = r_i(t)$ and $\mathfrak{m}_{i+1}(t) = \mathfrak{m}_i(t)$;

(v) for all $t \in T_{i+1} \setminus (T_i \setminus Des(chl(Fr_i)))$, $r_{i+1}(t) \in En_{\mathfrak{m}_{i+1}}(t)$;

Constraint (i) requires for the frontier of the hierarchical word to belong to the frontier of the source configuration tree. This enforces the synchronous communication model amongst siblings. Note that the hierarchical word described in Example 12 and depicted in Figure 3.1 is not a valid word for CSA$^{\lightning}$, since in ξ_2 the frontier consists in non-sibling components. Constraint (ii) requires that each component belonging to the interrupting components set In_i is actually able to perform an interrupting transition, i.e. the *interruptible* predicate holds in that component in the current configuration tree. Constraint (iii) enforces synchronization amongst siblings by requiring that each component belonging to the frontier can perform a transition labelled with a common communication symbol γ. Moreover, constraint (iii) defines the next configuration tree's labelling for the nodes belonging to the frontier accordingly to the transition function. In particular, for components t being in the interrupting set or being labelled by locations, the next configuration tree's labelling $r_{i+1}(t)$ must be one of the values in $\delta_{\mathfrak{m}_i(t)}(r_i(t), v_i(t), \gamma)$. For box-labelled components belonging to the frontier but not to the interrupting set, the next configuration tree's labelling $r_{i+1}(t)$ is required to be in $\delta_{\mathfrak{m}_i(t)}\left(\left(r_i(t), \prod_{t' \in chl(t)} r_i(t')\right), v_i(t), \gamma\right)$.

Example 15 (On CSA$^{\lightning}$ computations)**.** This example shows different runs of the CSA$^{\lightning}$ $\mathscr{A}$ described in Example 14 over several interrupting hierarchical words constructed over the symbols in Figure 3.3. Consider the hierarchical word $\xi^1 = (\xi_a)^\omega$. It produces the infinite run $\pi^1 = \langle T = \{\varepsilon\}, r = \{(\varepsilon, S)\}, \mathfrak{m} = \{(\varepsilon, \mathscr{A})\}\rangle, (\langle T = \{\varepsilon\}, r = \{(\varepsilon, P)\}, \mathfrak{m} = \{(\varepsilon, \mathscr{A})\}\rangle)^\omega$.

The interrupting hierarchical word $\xi^2 = \xi_a, (\xi_b)^\omega$ induces the run $\pi^2 = \langle T = \{\varepsilon\}, r = \{(\varepsilon, S)\}, \mathfrak{m} = \{(\varepsilon, \mathscr{A})\}\rangle, \langle T = \{\varepsilon\}, r = \{(\varepsilon, P)\}, \mathfrak{m} = \{(\varepsilon, \mathscr{A})\}\rangle, (\langle T = \{\varepsilon\}, r = \{(\varepsilon, Q)\}, \mathfrak{m} = \{(\varepsilon, \mathscr{A})\}\rangle)^\omega$.

The word $\xi^3 = \xi_b, (\xi_c, \xi_d)^\omega$ induces the run π^3 in which the call to the box B never returns. Formally, the run $\pi^3 = \langle T = \{\varepsilon\}, r = \{(\varepsilon, S)\}, \mathfrak{m} = \{(\varepsilon, \mathscr{A})\}\rangle, \langle T = \{\varepsilon, 1, 2\}, r = \{(\varepsilon, B),(1, R),(2, R)\}, \mathfrak{m} = \{(\varepsilon, \mathscr{A}), (1, \mathscr{B}), (1, \mathscr{B})\}\rangle, (\langle T = \{\varepsilon, 1, 2\}, r = \{(\varepsilon, B),(1, X),(2, Y)\}, \mathfrak{m} = \{(\varepsilon, \mathscr{A}), (1, \mathscr{B}), (1, \mathscr{B})\}\rangle, \langle T = \{\varepsilon, 1, 2\}, r = \{(\varepsilon, B), (1, X), (2, Y)\}, \mathfrak{m} = \{(\varepsilon, \mathscr{A}), (1,\mathscr{B}), (1,\mathscr{B})\}\rangle)^\omega$.

The word $\xi^4 = \xi_b, \xi_c^2, \xi_e, (\xi_b)^\omega$ induces the run π^4 in which there is a synchronous return from the call to the box B, while the word $\xi^5 = \xi_b, \xi_c, \xi_f, (\xi_a)^\omega$ induces the run π^5 in which there is a return by interrupt from the call to B. For the sake of brevity, π^4 and π^5 are not described formally. All of the runs in this example are depicted in Figure 3.4, in which every node t of a configuration tree T_i is labelled with $r_i(t)$ and furtherly decorated with $\mathfrak{m}_i(t)$, and consecutive configuration trees in a run are linked by arrows decorated with the corresponding hierarchical word symbol inducing the computation step.

Before formalizing the notion of acceptance for an infinite run, it is necessary to introduce some preliminary definitions.

Definition 20 (*Still* points in an infinite run [24])**.** Given a run $\pi = \prod_{j\geq 0}\langle T_j, r_j, \mathfrak{m}_j\rangle$, a point $t \in \mathbb{N}^*$ is said to be *still* in π if, starting from a certain instant i onwards, it occurs in each configuration tree $T_{k\geq i}$, and is never involved in a transition. Formally, the set of *still* points in π is defined as follows:

$$Still(\pi) \triangleq \{t \in \mathbb{N}^* \mid \exists i \in [0, |\pi|).\forall j \in [i, |\pi|).t \in T_j \setminus Fr_j\}.$$

Definition 21 (*Continuous* points in an infinite run [24])**.** Given a run $\pi = \prod_{j\geq 0}\langle T_j, r_j, \mathfrak{m}_j\rangle$, a point $t \in \mathbb{N}^*$ is *continuous* in π if it either is the root ε or is a child of a *still* point. Formally, the set of *continuous* points in π is defined as $Cont(\pi) \triangleq \{\varepsilon\}\cup\{t = t'\cdot i \in \mathbb{N}^* : t' \in Still(\pi)\}$.

A continuous point is eventually associated with a component that never returns, since its father, from a certain instant onwards, never performs transitions. A still point is always continuous, but the opposite is not necessarily true. In run π_3 discussed in Example 15, for example, nodes 1 and 2 are continuous but are not still. Furthermore, notice that the initial component – i.e. the one being at the root of the hierarchy – is always continuous as it is required to run on infinite hierarchical words and cannot return.

Definition 22 (*Recurrent* points in an infinite run [24])**.** Given a run $\pi = \prod_{j\geq 0}\langle T_j, r_j, \mathfrak{m}_j\rangle$, a point $t \in \mathbb{N}^*$ is *recurrent* in π if it is continuous but not still. Formally, the set of *recurrent* points in π is defined as $Rec(\pi) \triangleq Cont(\pi) \setminus Still(\pi)$.

The set of control states occurring infinitely often at node t along an infinite computation π is defined as $Inf_\pi(t) \triangleq \{q \in Q \mid \exists^\infty j \in \mathbb{N}.r_j(t) = q \wedge t \in T_j\}$. Note that $Inf_\pi(t)$ is always a singleton when $t \in Still(\pi)$.

Definition 23 (Acceptance of an infinite CSA$^\xi$ run [24])**.** An infinite run π of a CSA$^\xi$ is accepting if the following conditions are fulfilled:

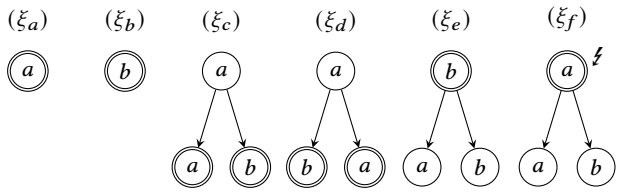


Figure 3.3: Hierarchical word symbols

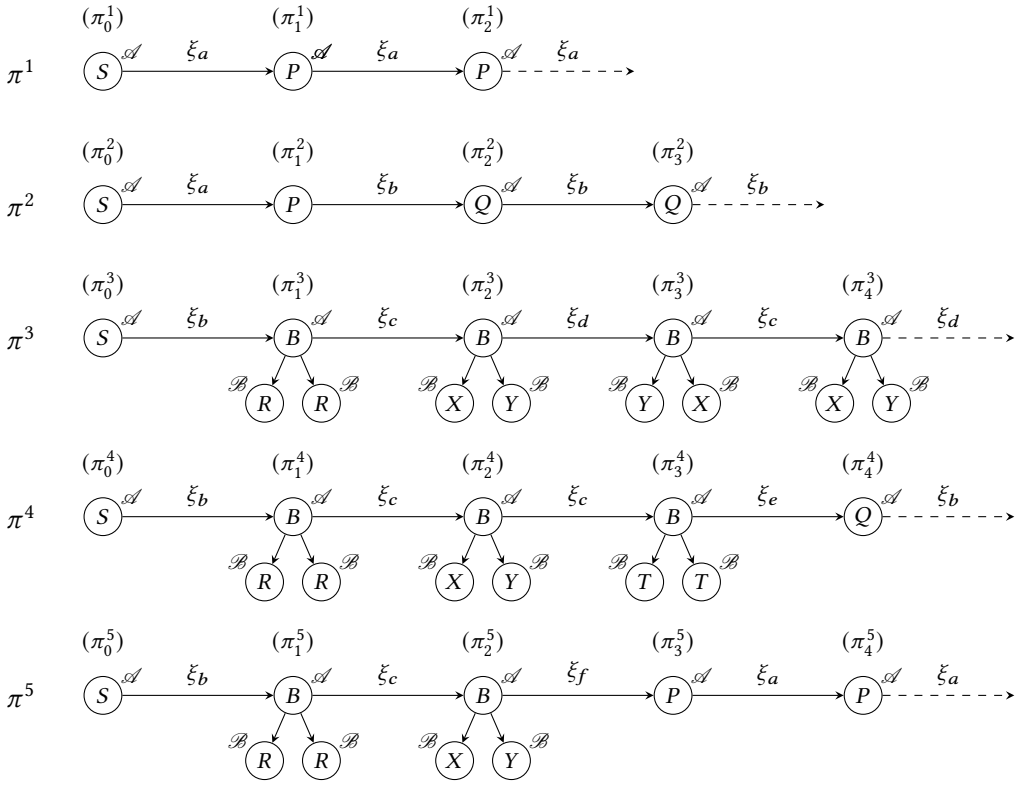


Figure 3.4: Runs on the hierarchical words in Example 15

(i) for each recurrent node $t \in Rec(\pi)$ and for each accepting set $R_i \in R_{\mathfrak{m}(t)}$, it holds that $Inf_\pi(t) \cap R_i \neq \emptyset$;

(ii) for each still node $t \in Still(\pi)$, it holds that $Inf_\pi(t) \subseteq F_{\mathfrak{m}(t)}$.

Note that, in a CSA$^{\lightning}$ $\mathscr{A}$, the recurrent acceptance sets in $R_{\mathscr{A}}$, correspond to a generalized Büchi condition [3, § 4.3.4] on recurrent nodes. As usual, $\mathcal{L}(\mathscr{A})$ denotes the set of all interrupting hierarchical words accepted by $\mathscr{A}$.

Example 16 (On the acceptance of CSA$^{\lightning}$ runs)**.** Consider the CSA$^{\lightning}$ $\mathscr{A}$ detailed in Example 14 and its runs π^i, $i \in \{1, \ldots, 5\}$ discussed in Example 15. Recall that $F_{\mathscr{A}} = \{B\}$, $R_{\mathscr{A}} = \{\{Q\}\}$, $F_{\mathscr{B}} = \emptyset$; $R_{\mathscr{A}} = \{\{Y\}\}$. In π^1 there are no still points and ε is a continuous and recurrent point. Since the only state being visited infinitely often in ε is P and the only recurring accepting set is $\{Q\}$, π^1 rejects the interrupting hierarchical word ξ^1. Conversely, π^2 accepts ξ^2 since $Inf_{\pi^2}(\varepsilon) = \{Q\}$. In the run π^3, node ε is still and continuous (and not recurrent), since from π^3_1 onwards it is never involved in any transition. Points 1 and 2, being children of a still node, are by definition continuous and, since they are infinitely often involved in transitions, recurrent. Since $Inf_{\pi^3}(\varepsilon) = \{B\}$ is included in $F_{\mathscr{A}}$ and, for all $R \in R_{\mathscr{B}}$, the intersection $Inf_{\pi^3}(n) \cap R = \{Y\}$, with $n \in \{1, 2\}$, π^3 is an accepting run. Run π^4 has no still nodes and $Cont(\pi^4) = Rec(\pi^4) = \{\varepsilon\}$, and, much like π^1, is an accepting run since for each $R \in R_{\mathscr{A}}$ $Inf_{\pi^4}(\varepsilon) \cap R = \{Q\}$. On the contrary, π^5, which similarly has no still nodes and $Cont(\pi^4) = Rec(\pi^4) = \{\varepsilon\}$, is a rejecting run since there exists a $R \in R_{\mathscr{A}}$ such that $Inf_{\pi^5}(\varepsilon) \cap R = \emptyset$.

As defined in this section, CSA$^{\lightning}$ allow for modelling unrestricted recursive systems, but HLTL$^{\lightning}$ is not able to predicate over unbounded hierarchical structures. For this reason, as in [24], in what follows the focus is restricted on a subclass of CSA$^{\lightning}$ allowing only for hierarchies of bounded depth. This subclass is called *Communicating Hierarchical Automata with Interrupts* (CHA$^{\lightning}$) and is defined using stratification of calls as follows.

Definition 24 (Communicating Hierarchical Automata with Interrupts (CHA$^{\lightning}$))**.** The class of Communicating Hierarchical Automata with Interrupts over Σ, Γ, Q, B, is defined as follows:

$$CHA(\Sigma, \Gamma, Q, B) = \bigcup_{i \geq 0} CHA_i(\Sigma, \Gamma, Q, B) \subset CSA(\Sigma, \Gamma, Q, B)$$

where:

(i) $CHA_0(\Sigma, \Gamma, Q, B) \triangleq CSA(\Sigma, \Gamma, Q, \emptyset)$;

(ii) $CHA_{i+1}(\Sigma, \Gamma, Q, B) = \left\{ \mathscr{A} \;\middle|\; rng(\beta_{\mathscr{A}}) \subseteq CHA_i(\Sigma, \Gamma, Q, B)^+ \right\}$;

Intuitively, (i) is the base case of the inductive definition and defines the automata having depth 0 as the ones containing no box. (ii) inductively defined the automata of depth $i > 0$ as the ones having exclusively boxes refined only by automata having depth strictly lower than i.

Algorithm 1 CHA emptiness

signature EMPTINESS : $CHA^+ \rightarrow_{\overline{\mathscr{A}}} \text{InOut}(\overline{\mathscr{A}})$
1: **function** EMPTINESS($\overline{\mathscr{A}}$)
2: **return** $reach(prod(\text{UNBOX}(\overline{\mathscr{A}})))$;
3: **end function**

signature UNBOX $CHA^+ \rightarrow CFA^+$
1: **function** UNBOX($\overline{\mathscr{A}}$)
2: $\overline{\mathscr{A}'} \leftarrow \varepsilon$
3: **for** $0 \leq i < |\overline{\mathscr{A}}|$ **do**
4: $\mathscr{A}' \leftarrow \overline{\mathscr{A}}_i$
5: **for all** $b \in B_{\mathscr{A}'}$ **do**
6: $I \leftarrow \text{EMPTINESS}(\beta_{\mathscr{A}'}(b))$
7: $Q_{\mathscr{A}'} \leftarrow Q_{\mathscr{A}'} \cup (\{b\} \times \text{I})$ ▷ add *summary* states
8: $B_{\mathscr{A}'} \leftarrow B_{\mathscr{A}'} \setminus \{b\}$ ▷ remove b from the set of boxes
9: $\beta_{\mathscr{A}'} \leftarrow \delta_{\mathscr{A}'} \restriction B_{\mathscr{A}'}$ ▷ restrict the box refinement function
10: $F_{\mathscr{A}'} \leftarrow \text{acc}(F_{\mathscr{A}'}, b, I)$ ▷ suitably enrich accepting states
11: $R_{\mathscr{A}'} \leftarrow \{\text{acc}(X, b, I) \mid X \in R_{\mathscr{A}'}\}$ ▷ suitably enrich accepting states
12: $\delta_{\mathscr{A}'} \leftarrow \text{trn}(Q_{\mathscr{A}'}, \beta_{\mathscr{A}'}, \delta_{\mathscr{A}'}, b, I)$ ▷ update the transition function
13: **end for**
14: $\overline{\mathscr{A}'} \leftarrow \overline{\mathscr{A}'} \cdot \mathscr{A}'$
15: **end for**
16: **return** $\overline{\mathscr{A}'}$
17: **end function**

3.3 Deciding CHA$^{\lightning}$ emptiness

This section provides a decision procedure to compute the emptiness of the language accepted by the parallel composition of a sequence $\overline{\mathscr{A}}$ of CHA$^{\lightning}$. The proposed procedure is an extension of the one proposed in [24] for CHA and adapts the bottom-up summarization technique well-known in the context of hierarchical machines. The procedure is described in Algorithm 1. The algorithm takes as input a non-empty sequence of CHA$^{\lightning}$ $\overline{\mathscr{A}} = \{\mathscr{A}_1, \ldots, \mathscr{A}_k\}$ and returns a set $\text{EMPTINESS}(\overline{\mathscr{A}}) \subseteq \text{InOut}(\overline{\mathscr{A}}) \triangleq \{(\overline{s}, \overline{t}) \in L^+ \times (\{\top\} \cup L^+) \mid \forall i.\overline{s}_i \in En_{\mathscr{A}_i} \wedge \overline{t} = \top \vee \forall i.\overline{t}_i \in Ex_{\mathscr{A}_i}\}$, with the following semantics: $(\overline{s}, \overline{t}) \in \text{EMPTINESS}(\overline{\mathscr{A}})$ iff there exists a sequence of k computations $\overline{\pi}$ such that each $\overline{\pi}_i$ is accepting for $\overline{\mathscr{A}}_i$, synchronizes on actions taken at the higher lever of the hierarchy, and:

(i) $\overline{s}_i = r_0^i(\varepsilon)$ in the first configuration tree in the run $\overline{\pi}_i$, for all i;

(ii) $\overline{t} = \top$ if each run in $\overline{\pi}_i$ is infinite or if $last(\overline{\pi}_i) \notin Ex_{\mathscr{A}_i}$ for some i, and $\overline{t} = last(\overline{\pi}_i)$, otherwise.

The main intuition is that each of the pairs $(\overline{s}, \overline{t})$ returned by the algorithm is witness of a sequence of accepting runs, one for each automaton given in input. The emptiness computation proceeds as follows. Firstly, each of the automaton in the input sequence is "unboxed", i.e. the boxes are removed and substituted with appropriate *summaries* between entries and exits. This unboxing operation produces a sequence of flat automata whose parallel composition computed by the function prod is equivalent to the parallel composition of the automata in $\overline{\mathscr{A}}$. The flat automaton $\mathscr{A}_F \triangleq \mathsf{prod}(\mathsf{unbox}(\overline{\mathscr{A}}))$, which as usual has as states tuples of states of the flattened automata produced by unbox, is then passed as input to the function reach, which computes the set of pairs $(\overline{s}, \overline{t}) \in \mathrm{InOut}(\overline{\mathscr{A}})$.

The unboxing procedure, taking as input a sequence of CHA^{ℓ} $\overline{\mathscr{A}}$ and returning a sequence of flat automata $\overline{\mathscr{A}'}$, is described as follows. For each input automaton $\mathscr{A}'$ and for each box $b \in B_{\mathscr{A}'}$, the algorithm computes – on line 6 – the emptiness result I for the sequence of CHA^{ℓ} in $\beta_{\mathscr{A}'}$. After that, in line 7, the set of summary states $b \times \{I\}$ is added to the set of states $Q_{\mathscr{A}'}$. In line 8, b is removed from the set of boxes and, after that, the box refinement function $\beta_{\mathscr{A}'}$ is properly restricted. Notice that, at this point, b is still a state but is no longer a box. Lines 10 and 11 properly enrich the acceptance conditions to account for the newly-introduced summary states. The intuition is that, if the box b is an accepting state in the CHA^{ℓ}, then all the corresponding summary states of the form $(b, (\overline{s}, \overline{t}))$ are to be accepting as well. To ensure so, the function acc, defined as follows, is used. $\mathrm{acc}(X, b, I)$ is equal to X if $b \notin X$ and to $X \cup (\{b\} \times I)$ otherwise. Finally, on line 12, the transition relation is properly updated to include edges to and from the newly-introduced summary states. The new transition relation is obtained by applying the function trn defined as follows. $\mathrm{trn}(Q, \beta, \delta, b, I)$ is the union of the following sets of edges:

(i) $\{(s, \sigma, \gamma, t) \in \delta \mid s \neq b, s \neq (b, \langle ex_1, \ldots, ex_{|\beta(b)|}\rangle), t \neq (b, \langle en_1, \ldots, en_{|\beta(b)|}\rangle)\}$ is the set of original edges not involving the box b;

(ii) $\{(s, \sigma, \gamma, (b, (t, w)) \mid (t, w) \in I \wedge (s, \sigma, \gamma, (b, t)) \in \delta\}$ is the set of edges replacing those in δ entering b;

(iii) $\{((b, (w, s)), \sigma, \gamma, t) \mid (w, s) \in I \wedge s \neq \top \wedge ((b, s), \sigma, \gamma, t) \in \delta\}$ is the set of edges replacing those in δ exiting b from reachable exits;

(iv) $\{(s, \sigma, \gamma, b) \mid (s, \sigma, \gamma, (b, t)) \in \delta\}$ is the set of edges entering b (for each edge entering the structured state (b, t) in the original relation, a new edge entering the location b is introduced);

(v) $\{(s, \sigma, \gamma, t) \in \delta \mid s = b\}$ is the set of original interrupting edges exiting the box b.

3.4 Satisfiability of $\text{HLTL}^{\lightning}_{\text{L}}$ over hierarchical computations

This section provides an automata-based decidability procedure for the satisfiability problem of the local fragment of $\text{HLTL}^{\lightning}$ interpreted over the class of $\text{CHA}^{\lightning}$ computations. The proposed procedure is an extension of the one devised in [24] and grounds, like the latter, on an extension of the classic approach that, given an LTL formula ϕ, produces a non-deterministic generalized Büchi automaton $\mathscr{A}_\phi$ accepting the computations satisfying ϕ. This approach, proposed by Vardi and Wolper in [20], devised the topological structure for $\mathscr{A}_\phi$ by connecting sets of *locally-consistent* subformulae of ϕ, often referred to as *atoms*, in a suitable way. The intuition is that each atom takes care of guaranteeing local consistency, while the connections amongst them take care of the modal semantics of temporal operators. The generalized Büchi automaton built on this topological structure, with properly-defined acceptance conditions, is the automaton whose language is the set of all computations satisfying ϕ.

In this section, the intuition above is generalized in a non-trivial way, by extending the notion of atom along the additional directions related to hierarchy and concurrency, in order to synthesize a $\text{CHA}^{\lightning}$ accepting the interrupting hierarchical words satisfying a $\text{HLTL}^{\lightning}_{\text{L}}$ formula.

In what follows, the focus is restricted on $\text{HLTL}^{\lightning}_{\text{L}}$ formulae in negation normal form (see Definition 15), thus the subscript l for all temporal operators is dropped for the sake of notation.

Firstly, the notions of *closure* of a $\text{HLTL}^{\lightning}_{\text{L}}$ formula φ and atom over a set of $\text{HLTL}^{\lightning}_{\text{L}}$ formulae are defined as follows.

Definition 25 (Closure set of a $\text{HLTL}^{\lightning}_{\text{L}}$ formula [24])**.** Given $\varphi \in \text{HLTL}^{\lightning}_{\text{L}}$, its *closure* $\text{cls}(\varphi)$ is the smallest set satisfying the following properties:

(i) $\varphi \in \text{cls}(\varphi)$;

(ii) if $\texttt{Op}\psi \in \text{cls}(\varphi)$, with $\texttt{Op} \in \{\neg, \texttt{X}, \texttt{X}_{\lightning}\}$, then $\psi \in \text{cls}(\varphi)$;

(iii) if $\texttt{X}_{\lightning}\,\psi \in \text{cls}(\varphi)$, then $\texttt{X}\,\psi,\ \texttt{X}_{\lightning}\,\top,\ \downarrow_1 \top \in \text{cls}(\varphi)$;

(iv) if $\texttt{X}\,\psi \in \text{cls}(\varphi)$, then $\texttt{X}\,\top \in \text{cls}(\varphi)$;

(v) if $\psi_1 \texttt{Op} \psi_2 \in \text{cls}(\varphi)$, with $\texttt{Op} \in \{\wedge, \vee\}$, then $\psi_1, \psi_2 \in \text{cls}(\varphi)$;

(vi) if $\psi_1\,\texttt{U}\,\psi_2 \in \text{cls}(\varphi)$ (resp. $\psi_1\,\texttt{R}\,\psi_2 \in \text{cls}(\varphi)$), then $\psi_2 \vee (\psi_1 \wedge \texttt{X}(\psi_1\,\texttt{U}\,\psi_2)) \in \text{cls}(\varphi)$ (resp. $\psi_2 \wedge (\psi_1 \vee \texttt{X}(\psi_1\,\texttt{R}\,\psi_2)) \in \text{cls}(\varphi)$).

Definition 26 ($\text{HLTL}^{\lightning}_{\text{L}}$ atom [24])**.** An atom over a set of $\text{HLTL}^{\lightning}_{\text{L}}$ formulae $C \subseteq \text{HLTL}^{\lightning}_{\text{L}}$ is a set $\alpha \subseteq C \cup \{en, ex\} \cup \neg\{en, ex\}$ satisfying the following closure rules:

(i) $\bot \notin \alpha$;

(ii) $p \in \alpha$ iff $\neg p \notin \alpha$ for all $p \in \{en, ex\} \cup (\mathcal{P} \cap C)$;

(iii) if $\phi \in \alpha$ (resp. $\neg\phi \in \alpha$), then $\neg\phi \notin \alpha$ (resp. $\phi \notin \alpha$);

(iv) if $en \in \alpha$ (resp. $ex \in \alpha$) then $ex \notin \alpha$ (resp. $en \notin \alpha$) and $\downarrow_i(\phi) \notin \alpha$ for any $i \in \mathbb{N}^+$ and $\phi \in \mathrm{HLTL}_{\mathrm{L}}^{f}$;

(v) if $ex \in \alpha$, then $\mathtt{X}\,\phi \notin \alpha$ for any $\phi \in \mathrm{HLTL}_{\mathrm{L}}^{f}$;

(vi) if $\phi_1 \wedge \phi_2 \in \alpha$ (resp. $\phi_1 \vee \phi_2 \in \alpha$), then $\phi_1, \phi_2 \in \alpha$ (resp. $\phi_1 \in \alpha$ or $\phi_2 \in \alpha$);

(vii) if $\phi_1\,\mathtt{U}\,\phi_2 \in \alpha$ (resp. $\phi_1\,\mathtt{R}\,\phi_2 \in \alpha$) then $\phi_2 \vee (\phi_1 \wedge \mathtt{X}(\phi_1\,\mathtt{U}\,\phi_2)) \in \alpha$ (resp. $\phi_2 \wedge (\phi_1 \vee \mathtt{X}(\phi_1\,\mathtt{U}\,\phi_2)) \in \alpha$).

The set of all atoms over $C \subseteq \mathrm{HLTL}_{\mathrm{L}}^{f}$ is denoted by $\mathrm{Atm}(C)$.Moreover, the following notation is introduced: $\mathrm{Atm}^{p}(C) \triangleq \{\alpha \in \mathrm{Atm}(C) \mid p \in \alpha\}$ is the set of all atoms containing the p, with $p \in \{en, ex\}$; $\mathrm{Atm}^{\rightarrow}(C) \triangleq \left\{\alpha \in \mathrm{Atm}(C) \mid \exists \phi \in \mathrm{HLTL}_{\mathrm{L}}^{f}.\, \rightarrow \phi \in \alpha\right\}$ is the set of all atoms requiring a right sibling; $\mathrm{Atm}^{F}(C) \triangleq \{\alpha \in \mathrm{Atm}(C) \mid \neg\,\mathtt{X}\,\top \in \alpha\}$ is the set of all final atoms, i.e. those not allowing for the existence of a successor; $\mathrm{Atm}^{\phi_1\,\mathtt{U}\,\phi_2}(C) \triangleq \{\alpha \in \mathrm{Atm}(C) \mid \phi_1\,\mathtt{U}\,\phi_2 \in \alpha \Rightarrow \phi_2 \in \alpha\}$ is the set of all atoms that locally satisfy the until subformula $\phi_1\,\mathtt{U}\,\phi_2$.

An atom α may represent a state requiring refinement by one or more automata. For example, an atom α such that $\downarrow_2(p) \in \alpha$ requires refinement by at least two automata and, in particular, requires for p to hold in its second child. To determine the formulae required to hold in each of the possibly many automata required to refine an atom, a *contextualization* function cnt is introduced and defined as follows.

Definition 27 (Contextualization function [24])**.** Given an $\mathrm{HLTL}_{\mathrm{L}}^{f}$ atom α, its contextualization $\mathrm{cnt}(\alpha)$ is a set of pairs in $\mathbb{N} \times \mathrm{HLTL}_{\mathrm{L}}^{f}$ with the following semantics: $(n, \psi) \in \mathrm{cnt}(\alpha)$ iff the $\mathrm{HLTL}_{\mathrm{L}}^{f}$ formula ψ is required by α to hold in its n-th child. More precisely, $\mathrm{cnt} : Atm \rightarrow 2^{\mathbb{N} \times \mathrm{HLTL}_{\mathrm{L}}^{f}}$ and $\mathrm{cnt}(\alpha)$ is the smallest set such that the following conditions hold:

(i) $(i, \phi) \in \mathrm{cnt}(\alpha)$ for all $\downarrow_i(\phi) \in \alpha$;

(ii) if $(i, \mathtt{Op}\phi) \in \mathrm{cnt}(\alpha)$, with $\mathtt{Op} \in \{\neg, \mathtt{X}, \mathtt{X}_f\}$ then $(i, \phi) \in \mathrm{cnt}(\alpha)$;

(iii) if $(i, \mathtt{X}_f\,\psi) \in \mathrm{cnt}(\alpha)$, then $(i, \mathtt{X}\,\psi)$, $(i, \mathtt{X}_f\,\top)$, $(i, \downarrow_1 \top) \in \mathrm{cnt}(\alpha)$;

(iv) if $(i, \mathtt{X}\,\psi) \in \mathrm{cnt}(\alpha)$, then $(i, \mathtt{X}\,\top) \in \mathrm{cnt}(\alpha)$;

(v) if $(i, \phi_1 \mathtt{Op} \phi_2) \in \mathrm{cnt}(\alpha)$, with $\mathtt{Op} \in \{\wedge, \vee\}$, then $(i, \phi_1), (i, \phi_2) \in \mathrm{cnt}(\alpha)$;

(vi) if $(i, \phi_1\,\mathtt{U}\,\phi_2) \in \mathrm{cnt}(\alpha)$ (resp. $(i, \phi_1\,\mathtt{R}\,\phi_2) \in \mathrm{cnt}(\alpha)$), then $(i, \phi_2 \vee (\phi_1 \wedge \mathtt{X}(\phi_1\,\mathtt{U}\,\phi_2))) \in \mathrm{cnt}(\alpha)$ (resp. $(i, \phi_2 \wedge (\phi_1 \vee \mathtt{X}(\phi_1\,\mathtt{R}\,\phi_2))) \in \mathrm{cnt}(\alpha)$);

(vii) if $(i, \leftarrow \phi) \in \mathrm{cnt}(\alpha)$ and $i > 1$ (resp. $(i, \rightarrow \phi) \in \mathrm{cnt}(\alpha)$) then $(i - 1, \phi) \in \mathrm{cnt}(\alpha)$ (resp. $(i + 1, \phi) \in \mathrm{cnt}(\alpha)$).

Note that the contextualization of an atom associated with a location is the empty set. Given an atom's contextualization, it is immediate to determine the maximum number of parallel automata over which it may predicate by means of the maxind function defined as follows: $\mathsf{maxind}(\alpha) \triangleq \max\{n \in \mathbb{N} | \exists \phi \in \mathrm{HLTL}^{f}_{\mathrm{L}}.(n, \phi) \in \mathsf{cnt}(\alpha)\}$. Atoms associated with boxes have strictly positive maxind, while the ones associated with locations have $\mathsf{maxind} = 0$.

The contextualization of an atom also allows for the computation of the closure and the set of atoms associated with its required i-th child, by means of the functions cls and atm defined as follows.

Definition 28 (Closure for the i-th child [24])**.** The function $\mathsf{cls} : \mathsf{Atm}(C) \times \mathbb{N}^{+} \mapsto 2^{\mathrm{HLTL}^{f}_{\mathrm{L}}}$ takes as input an atom α and a positive natural number i and collects all the $\mathrm{HLTL}^{f}_{\mathrm{L}}$ formulae ϕ in $\mathsf{cnt}(\alpha)$ that are required to hold in the i-th child. More precisely, $\mathsf{cls}(\alpha, i) = \{\phi \mid (i, \phi) \in \mathsf{cnt}(\alpha)\}$.

Definition 29 (Atoms for the i-th child [24])**.** Given an atom α and a positive natural number i, $\mathsf{atm}(\alpha, i)$ collects all the atoms α' in $\mathsf{Atm}(\mathsf{cls}(\alpha, i))$.

Recall that CHA^{f} semantics require that siblings execute synchronously in parallel (see Definition 19). When synthesizing lower-level automata, the problem of enforcing synchronization amongst siblings arises. The required synchronization can be obtained by means of a synchronization function syn which associates to each given atom α a set of couples in which the first element is a sequence of *synchronization-compatible* atoms and the second one is a sequence of possible target atoms. Each of these synchronization-compatible sequences represents a possible synchronization amongst sibling machines refining the box associated with α, when each of them is the state encoded by the corresponding atom in the sequence. Formally, the synchronization function is defined as follows.

Definition 30 (Synchronization function [24])**.** The synchronization function $\mathsf{syn} : \mathsf{Atm} \rightarrow_{\alpha} 2^{\bigcup_{i=1}^{\mathsf{maxind}(\alpha)} \mathsf{Atm}^i \times \mathsf{Atm}^i}$ is such that, for all $\langle \overline{\alpha}, \overline{\tau} \rangle \in \bigcup_{i=1}^{\mathsf{maxind}(\alpha)} \mathsf{Atm}^i \times \mathsf{Atm}^i$, $\langle \overline{\alpha}, \overline{\tau} \rangle \in \mathsf{syn}(\alpha)$ iff the following conditions hold:

(i) for all $1 \leq i \leq |\overline{\alpha}|$, $\overline{\alpha}_i, \overline{\tau}_i \in \mathsf{atm}(\alpha, i)$;

(ii) either $en \in \overline{\alpha}_i$ (resp. $ex \in \overline{\alpha}_i$) for all $1 \leq i \leq |\overline{\alpha}|$, or $en \notin \overline{\alpha}_i$ (resp. $ex \notin \overline{\alpha}_i$), for all $1 \leq i \leq |\overline{\alpha}|$;

(iii) if $\rightarrow \phi \in \overline{\alpha}_i$, then $\phi \in \overline{\alpha}_{i+1}$, for all $1 \leq i < |\overline{\alpha}|$;

(iv) if $\leftarrow \phi \in \overline{\alpha}_i$, then $\phi \in \overline{\alpha}_{i-1}$, for all $1 < i \leq |\overline{\alpha}|$;

(v) for all $1 \leq i < |\overline{\alpha}|$, $\neg \rightarrow \top \notin \overline{\alpha}_i$;

(vi) for all $1 < i \leq |\overline{\alpha}|$, $\neg \leftarrow \top \notin \overline{\alpha}_i$;

(vii) if $\neg \downarrow_i \top \in \alpha$ then $|\overline{\alpha}| < i$.

Constraint (ii) requires that a sibling in an entry state (resp. exit state) can only synchronize with siblings in entry states (resp. exit states). Constraints (iii) and (iv) are necessary to enforce consistency on siblings, i.e. a sibling requiring $\rightarrow(\phi)$ (resp. $\leftarrow(\phi)$) can synchronize only with a right (resp. left) sibling requiring ϕ. Constraints (v) and (vi) require that sequences are well-formed, i.e. all the elements but the last one cannot require the non-existence of a right sibling and all the elements but the first one require the non-existence of a left sibling. Finally, constraint (vii) guarantees that an atom containing $\neg\downarrow_i \top$ for some $i \in \mathbb{N}^+$ cannot have synchronization sequences of length i or more, since it cannot have more than i children.

An $\text{HLTL}^{f}_{\text{L}}$ atom, because of the possible disjunctive combinations of the operators $\rightarrow$, $\leftarrow$ and $\downarrow_i$, may require refinement by different numbers of parallel machines. Think, for example, of atoms containing formulae like $(\downarrow_1(p \wedge (\neg \rightarrow \top)) \vee (\downarrow_1(q \wedge (\rightarrow p))$. Such atoms represent boxes which can be refined by either 1 or 2 children. In this cases, the synchronization function returns sequences of different length. To disambiguate such cases, the following two-step procedure is used:

1. firstly, the atom is associated with the indices indicating all the possible lengths of the synchronization sequences associated with the corresponding box. This is achieved by means of the $\mathsf{indatm} : \mathsf{Atm} \mapsto 2^{\mathbb{N}}$, where $\mathsf{indatm}(\alpha) \triangleq \{|\overline{a}| \in \mathbb{N} \mid \overline{a} \in \mathsf{syn}(\alpha)\}$. Each pair in $\mathsf{Atm} \times_\alpha \mathsf{indatm}(\alpha)$ represents a possible refinement of the corresponding atom;

2. secondly, for each index $i \in \mathsf{indatm}(\alpha)$, the synchronization sequences of length i are collected by means of the function $\mathsf{syn} : \mathsf{Atm} \times_\alpha \mathsf{indatm}(\alpha) \mapsto_{(\alpha,i)} 2^{\mathsf{Atm}^i}$, where $\mathsf{syn}(\alpha, i) = \{\langle \overline{\alpha}, \overline{\tau} \rangle \in \mathsf{syn}(\alpha) \mid |\overline{\alpha}| = i\}$.

What remains is to determine, for each given $\text{HLTL}^{f}_{\text{L}}$ formula ϕ, the atoms associated with the top-level automaton. Intuitively, these top-level atoms cannot require for the existence of siblings, since the top-level automaton executes in isolation. Before formalizing the top-level atoms, the function seed-atom, associating to each $\text{HLTL}^{f}_{\text{L}}$ formula an atom, is introduced and defined as follows: $\mathsf{seed\text{-}atom} : \text{HLTL}^{f}_{\text{L}} \mapsto \mathsf{Atm}$ is such that $\mathsf{seed\text{-}atom}(\phi) = \{\downarrow_1(\phi), \neg\, en, \neg\, ex\}$. The $\mathsf{seed\text{-}atom}(\phi)$ associated to each formula ϕ has no deep meaning and is merely a technical tool necessary to guarantee that top-level atoms do not require for the existence of siblings, as explained below. It is now possible to associate with each $\text{HLTL}^{f}_{\text{L}}$ formula ϕ the set of top-level atoms by means of the function $\mathsf{atm} : \text{HLTL}^{f}_{\text{L}} \mapsto_\phi 2^{\mathsf{Atm}(\mathsf{cls}(\phi))}$, which is defined as follows.

Definition 31 (Top-level atoms for a $\text{HLTL}^{f}_{\text{L}}$ formula [24])**.** Given a $\text{HLTL}^{f}_{\text{L}}$ formula ϕ, the set of initial atoms $\mathsf{atm}(\phi)$ is the set of all atoms $\alpha \in \mathsf{Atm}(\mathsf{cls}(\phi))$ such that the following conditions are satisfied:

(i) $\alpha \in \{\overline{\alpha}_1 \mid \langle \overline{\alpha}, \overline{\tau} \rangle \in \mathsf{syn}\,(\mathsf{seed\text{-}atom}(\phi)) \wedge |\overline{\alpha}| = 1\}$;

(ii) if $en \in \alpha$ then $\phi \in \alpha$, since, by $\text{HLTL}^{f}_{\text{L}}$ semantics (see Definition 14), ϕ is required to hold in the initial states;

Algorithm 2 $\text{HLTL}^{\lightning}_{\text{L}}$ to CHA translation

signature $\textsc{Aut} : \text{HLTL}^{\lightning}_{\text{L}} \to CHA$
1: **function** $\textsc{Aut}(\varphi)$
2: **return** $\textsc{Aut}_1(\{\varphi\}, \text{atm}(\varphi), 1)$; ▷ Synthesize top-level automaton
3: **end function**

signature $\textsc{Aut}_k\ 2^{\text{HLTL}^{\lightning}_{\text{L}}} \times \text{Atm}^k \times [1,k] \to CHA$
1: **function** $\textsc{Aut}_k(\Phi, \Gamma, i)$
2: $A \leftarrow \{\overline{\alpha}_i \in Atm : \langle \overline{\alpha}, \overline{\tau} \rangle \in \Gamma\}$
3: $Q \leftarrow \{(\alpha, d) \in A \times \mathbb{N} : d \in \text{indatm}(\alpha)\}$
4: $En \leftarrow \{(\alpha, 0) \in Q : \Phi \subseteq \alpha \in Atm^{en}\}$
5: $Ex \leftarrow \{(\alpha, 0) \in Q : \alpha \in Atm^{ex}\}$
6: $B \leftarrow \{(\_, d) \in Q : d > 0\}$
7: **for all** $b = (\alpha, d) \in B$ **do**
8: $\beta(b) \leftarrow \varepsilon$
9: **for** $1 \leq j \leq d$ **do**
10: $\beta(b) \leftarrow \beta(b) \cdot \textsc{Aut}_d(\{\psi | \downarrow_j \psi \in \alpha\}, syn(\alpha, d), j)$
11: **end for**
12: **end for**
13: $\delta \leftarrow trn(Q, \Gamma, \beta, i)$
14: $F \leftarrow \{(\alpha, \_) \in Q : \alpha \in Atm^F\}$
15: $R \leftarrow \{\{(\alpha, \_) \in Q : \alpha \in Atm^{\phi_1 \mathtt{U} \phi_2}\} : \exists \alpha \in A. \phi_1 \mathtt{U} \phi_2 \in \alpha\}$
16: **return** $\langle Q, En, Ex, B, \beta, \delta, F, R \rangle$
17: **end function**

(iii) $ex \notin \alpha$, since the top-level automaton is not allowed to terminate.

Condition (i) selects as top-level atoms only those that are contained in synchronization sequences of the seed-atom(ϕ) having length 1, i.e. not requiring for any sibling.

With the previously-defined technical tools in place, it is now possible to describe the proposed synthesis procedure, shown in Algorithm 2. The main procedure $\textsc{Aut}$ takes as input an $\text{HLTL}^{\lightning}_{\text{L}}$ formula and returns a $\text{CHA}^{\lightning}$ accepting the interrupting hierarchical words satisfying the formula. To do so, the auxiliary function $\textsc{Aut}_k : 2^{\text{HLTL}^{\lightning}_{\text{L}}} \times \text{Atm}^k \times [1,k] \to CHA$ is used. $\textsc{Aut}_k(\Phi, \Gamma, i)$ synthesizes the i-th child automaton is a sequence of k siblings executing synchronously in parallel, with Φ being the set of formulae required in the initial states and Γ being the set of synchronization symbols over which the automaton and its siblings need to synchronize. When synthesizing the top-level automaton, which works in isolation and has no siblings, for the formula φ (see line 2), the function $\textsc{Aut}_1$ is used, φ is the only required formula in the initial states and the automaton is allowed to synchronize on every possible initial atom in atm(φ). The synthesis of the $\text{CHA}^{\lightning}$ $\textsc{Aut}_k(\Phi, \Gamma, i)$ proceeds as follows. In line 2, the atoms for the automaton to be synthesized are collected by extracting the i-th atoms in the synchronization sequences in Γ. Then,

in line 3, the set of control states is constructed by computing $A \times_\alpha \mathsf{indatm}(\alpha)$. In line 4, entering locations are identified as the control states (α, d) satisfying the following conditions: (i) no refinement is required, i.e. $d = 0$; (ii) the corresponding atom represents an initial state, i.e. $en \in \alpha$; (iii) the required formulae Φ are satisfied, i.e. $\Phi \subseteq \alpha$. Similarly, in line 5, exiting locations are collected as the states whose corresponding atom is in Atm^{ex}. The set of boxes is then defined as the set of control states (α, d) requiring refinement by at least one child, i.e. having $d > 0$. In lines 7–12 the required refinements for each box are synthesized and the refinement function β is defined. In greater detail, for each box $b = (\alpha, d)$, and for each required j-th refinement of b, the corresponding automaton is synthesized by calling $\textsc{Aut}_d$ as in line 10. In this call, the set of required formulae in the initial state is the set of all ψ such that $\downarrow_j(\psi)$ is required in the box's atom α, the set of synchronization symbols is the set of all synchronization sequences having length equal to d, and the child to synthesize is the j-th one. In line 13, a call to the function trn defines the transition relation for the synthesized automaton according to the following rules:

(i) a state $q = (\alpha, d)$ with $\neg \mathtt{X} \top \in \alpha$, cannot have any outgoing transition;

(ii) a state $b = (\alpha, d)$ with $\neg \mathtt{X}_\ell \top \in \alpha$ cannot have outgoing interrupting transitions;

(iii) a state $b = (\alpha, d)$ with $\mathtt{X}_\ell \psi \in \alpha$ for some $\psi \in \mathrm{HLTL}^{\ell}_{\mathrm{L}}$ can only have outgoing interrupting transitions;

(iv) if a state q is allowed to have successors, i.e. it does not fall in case (i), then each target of its transitions must be such such that its associated atom contains all the subformulae ϕ such that $\mathtt{X}\,\phi$ or $\mathtt{X}_\ell\,\phi \in \alpha$;

(v) in an automaton being the i-th child refining a box, any source s associated to some state $q = (\alpha, d)$ can have an outgoing transition to some target t with input symbol σ and synchronization symbol $\overline{\alpha}$ iff it agrees with the atomic propositions in σ, $\overline{\alpha}_i = \alpha$, and $\overline{\tau}_i = \alpha_t$, with α_t being the atom corresponding to the target t.

Before formalizing the trn function, the set of sources and targets associated with a state $q = (\alpha, d)$, with the refinement function β, are defined respectively as follows:

$$src_\beta(q) = \begin{cases} \{q\}, & \text{if } d = 0; \\ \{q\}, \cup\{(q, ex_1, \ldots, ex_d) \mid \forall i \in \{1, \ldots, d\}.\, ex_i \in Ex_{\beta(q)_i}\} & \text{otherwise.} \end{cases}$$

$$trg_\beta(q) = \begin{cases} \{q\}, & \text{if } d = 0; \\ \{(q, en_1, \ldots, en_d) \mid \forall i \in \{1, \ldots, d\}.\, en_i \in En_{\beta(q)_i}\}, & \text{otherwise.} \end{cases}$$

With the definitions of sources and targets associated with a state in place, it is possible to formalize the function trn as follows: $\delta = \mathsf{trn}(Q, \Gamma, \beta, i)$ is such that, for each input symbol $\sigma \in \Sigma$ and synchronization symbol $\gamma = \langle \overline{\alpha}, \overline{\tau} \rangle \in \Gamma$, for all states $q = (\alpha, d) \in Q$

and sources $s \in src_\beta(q)$, it holds that

$$\delta(s,\sigma,\gamma) = \begin{cases} \left\{ \begin{array}{r} t \in trg_\beta(q') : q' = (\overline{\tau}_i, \_) \in Q \wedge \\ \forall\, \mathtt{X}\,\phi \in \alpha . \phi \in \overline{\tau}_i \end{array} \right\}, & \text{if } \neg\,\mathtt{X}\top \notin \alpha \text{ and } \neg\,\mathtt{X}_{\not\iota}\top \in \alpha \text{ and} \\ & \quad \overline{\alpha}_i = \alpha \text{ and } \sigma = \alpha \cap \mathcal{P} \text{ and} \\ & \quad (d > 0 \Rightarrow s \neq q); \\ \left\{ \begin{array}{r} t \in trg_\beta(q') : q' = (\overline{\tau}_i, \_) \in Q \wedge \\ \forall \mathtt{Op}\phi \in \alpha . \phi \in \overline{\tau}_i, \mathtt{Op} \in \{\mathtt{X}, \mathtt{X}_{\not\iota}\} \end{array} \right\}, & \text{if } \neg\,\mathtt{X}\top \notin \alpha \text{ and } \exists \phi .\, \mathtt{X}_{\not\iota}\,\phi \in \alpha \text{ and} \\ & \quad \overline{\alpha}_i = \alpha \text{ and } \sigma = \alpha \cap \mathcal{P} \text{ and } s = q; \\ \left\{ \begin{array}{r} t \in trg_\beta(q') : q' = (\overline{\tau}_i, \_) \in Q \wedge \\ \forall\, \mathtt{X}\,\phi \in \alpha . \phi \in \overline{\tau}_i \end{array} \right\}, & \text{if } \neg\,\mathtt{X}\top \notin \alpha \text{ and } \overline{\alpha}_i = \alpha \text{ and} \\ & \quad \sigma = \alpha \cap \mathcal{P}; \\ \varnothing, & \text{otherwise.} \end{cases}$$

The above case analysis is explained as follows.

The first case guarantees that a state whose corresponding atom does not explicitly forbid the existence of a successor ($\neg\,\mathtt{X}\top \notin \alpha$), but forbids the existence of interrupting successors ($\neg\,\mathtt{X}_{\not\iota}\top \in \alpha$), is only allowed to have non-interrupting transitions satisfying condition (v) in the previous informal description ($\overline{\alpha}_i = \alpha$, $\sigma = \alpha \cap \mathcal{P}$, and the target atom is forced to match $\overline{\tau}_i$). In this case, the target atoms must also contain every subformula ψ such that $\mathtt{X}\,\psi \in \alpha$ (notice that α cannot contain formulae of the form $\mathtt{X}_{\not\iota}\,\psi$ since $\neg\,\mathtt{X}_{\not\iota}\top \in \alpha$). The fact that only non-interrupting transitions are allowed is guaranteed by condition ($d > 0 \Rightarrow s \neq q$). This condition requires that, if the state is a box (i.e. $d > 0$), then the source cannot be non-structured state. Recall that, in CHA$^{\not\iota}$ (see Definition 16), interrupting transitions have as source the box alone, while non-interrupting synchronous return transitions must specify a box and a sequence of exiting states for the box's children, e.g. $(b, \overline{ex})$.

The second case guarantees that a state whose corresponding atom does not explicitly forbid the existence of a successor and contains some formula of the form $\mathtt{X}_{\not\iota}\,\phi$, is allowed to have only interrupting transitions satisfying, as in the previous case, condition (v). In this case the target atoms are required to contain every ψ such that $\mathtt{X}\,\psi \in \alpha$ or $\mathtt{X}_{\not\iota}\,\psi \in \alpha$. The fact that only interrupting transitions are allowed is guaranteed by condition $s = q$. Notice that there is no need to specify that s must be a box since the associated atom contains $\mathtt{X}_{\not\iota}\,\psi$ and, by closure rules (see Definition 25), must also contain $\downarrow_1 \top$ and thus is necessarily a box.

The third case is applied to states whose corresponding atoms do not explicitly deny the existence of a successor and neither forbid the existence of an interrupting successor nor require it. Such states are allowed to perform both interrupting and non-interrupting transitions satisfying the previously-discussed constraint (v) to states whose corresponding atoms contain every subformula ψ such that $\mathtt{X}\,\psi \in \alpha$ (notice that α cannot contain formulae of the form $\mathtt{X}_{\not\iota}\,\psi$, as it would have fallen in the previous case).

In every other case, i.e. when the existence of a successor is explicitly denied ($\neg\,\mathtt{X}\top \in \alpha$), or σ does not agree with the source atom on the atomic propositions ($\sigma \neq \alpha \cap \mathcal{P}$), or the synchronization sequence is inappropriate (i.e. $\overline{\alpha}_i \neq \alpha$), no transition is allowed.

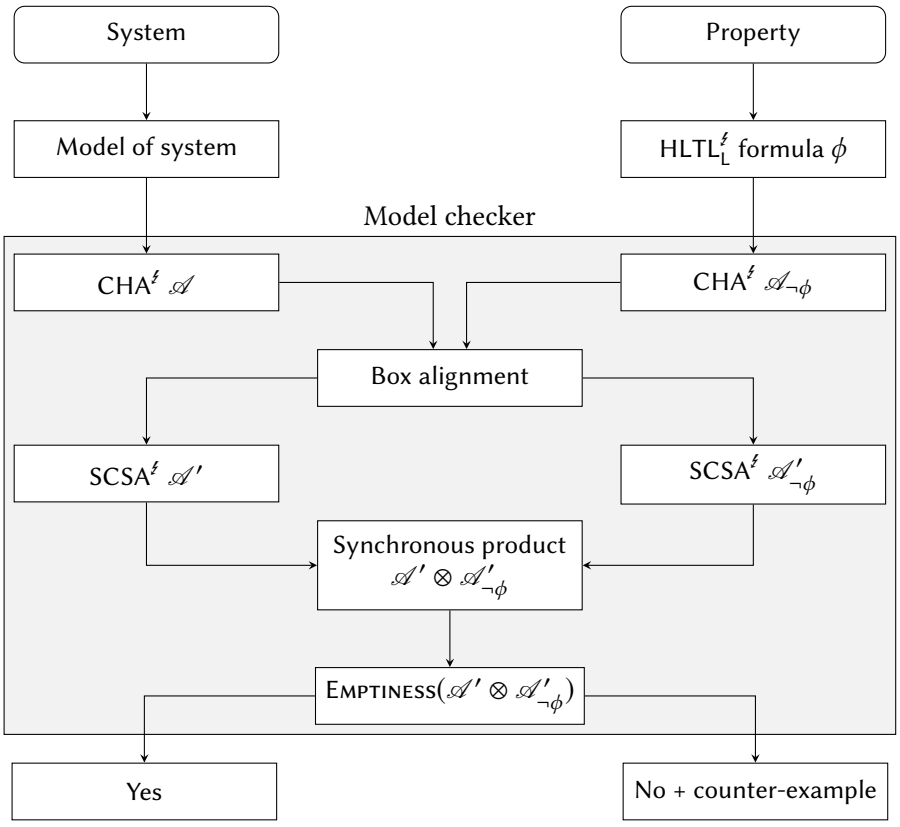


Figure 3.5: Overview of HLTL$^{\not\downarrow}_{\mathrm{L}}$ model checking

In lines 14 and 15 the acceptance conditions are defined. F is defined as the set of states associated with atoms not requiring a successor, while R contains, for each until subformula belonging to some atom in A, a set of atoms locally satisfying the until. These generalized Büchi conditions are necessary with until subformulae to reject computations in which the second until operand is never actually satisfied, as explained in [20] [3, §5.2].

With the automaton $\mathscr{A}_\phi = \textsc{Aut}(\phi)$ in place, solving the satisfiability problem for ϕ amounts to decide the emptiness of $\mathcal{L}(\mathscr{A}_\phi)$, which can be done as described in Section 3.3.

3.5 HLTL$^{\not\downarrow}_{\mathrm{L}}$ Model Checking

Given an HLTL$^{\not\downarrow}_{\mathrm{L}}$ formula ϕ over the set of atomic propositions $\mathcal{P}$ and a CHA$^{\not\downarrow}$ model $\mathscr{A} \in \mathrm{CHA}^{\not\downarrow}(2^{\mathcal{P}}, \Gamma, Q, B)$, the model checking problem consists in deciding whether every interrupting hierarchical word ξ accepted by $\mathscr{A}$ is such that of $\xi \vDash \phi$. The procedure described in this section and represented in Figure 3.5 is an extension of the classic, previously-cited, automata-based LTL model checking described in [20, 3].

The main intuition is to represent the negation of the HLTL$^{\not\downarrow}_{\mathrm{L}}$ formula ϕ as a CHA$^{\not\downarrow}$, which can be done as described in Section 3.4 and in Algorithm 2, then compute the synchronous product of the system's and the formula's automata, thus obtaining an

automaton whose emptiness can be checked to look for runs satisfying the negation of the property, i.e. to disprove the fact that the property holds in every system's run.

With the synthesis and the emptiness problems already addressed in the previous sections, what remains is to properly define the synchronous product of two CHA$^{\lightning}$. In what follows, the synchronous product is defined for *box-compatible* automata. Then, since the system's and the formula's automata might not be *box-compatible*, a *box alignment* procedure to make them *box-compatible* without altering the model checking's result is defined.

Definition 32 (Box-compatible CSA$^{\lightning}$)**.** Given two CSA$^{\lightning}$ $\mathscr{A}_1 \in \text{CSA}^{\lightning}(\Sigma, \Gamma_1, Q_1, B_1)$ and $\mathscr{A}_2 \in \text{CSA}^{\lightning}(\Sigma, \Gamma_2, Q_2, B_2)$ over the same set of input symbols Σ, $\mathscr{A}_1$ and $\mathscr{A}_2$ are *box-compatible*, in symbols $\mathscr{A}_1 \propto_b \mathscr{A}_2$, iff the following conditions hold:

(i) all the boxes in both automata are refined by the same number of machines, i.e. for every $(b_1, b_2) \in B_{\mathscr{A}_1} \times B_{\mathscr{A}_2}$, $|\beta_{\mathscr{A}_1}(b_1)| = |\beta_{\mathscr{A}_2}(b_2)|$;

(ii) for each pair $(b_1, b_2) \in B_{\mathscr{A}_1} \times B_{\mathscr{A}_2}$ and for each $i \in [0, |\beta_{\mathscr{A}_1}(b_1)| - 1]$, it holds that $\beta_{\mathscr{A}_1}(b_1)_i \propto_b \beta_{\mathscr{A}_2}(b_2)_i$.

Definition 33 (Synchronous product of CSA$^{\lightning}$)**.** Given two box-compatible automata $\mathscr{A}_1 \in \text{CSA}^{\lightning}(\Sigma, \Gamma_1, Q_1, B_1)$ and $\mathscr{A}_2 \in \text{CSA}^{\lightning}(\Sigma, \Gamma_2, Q_2, B_2)$, the synchronous product $\mathscr{A}_1 \otimes \mathscr{A}_2$ is the CSA$^{\lightning}$ defined as follows:

$$\mathscr{A}_1 \otimes \mathscr{A}_2 = \langle Q, En, Ex, B, \beta, \delta, F, R \rangle$$

where:

- $Q = Q_{\mathscr{A}_1} \times Q_{\mathscr{A}_2}$, $En = En_{\mathscr{A}_1} \times En_{\mathscr{A}_2}$, $Ex = Ex_{\mathscr{A}_1} \times Ex_{\mathscr{A}_2}$, $B = B_{\mathscr{A}_1} \times B_{\mathscr{A}_2}$;
- for all $b = (b_1, b_2) \in B$, $\beta(b) = \prod_{i=0}^{|\beta_{\mathscr{A}_1}(b_1)|-1} (\beta_{\mathscr{A}_1}(b_1)_i \otimes \beta_{\mathscr{A}_2}(b_2)_i)$;
- for all $(q_1, q_2) \in Src_{\mathscr{A}_1} \times Src_{\mathscr{A}_2}$, $\sigma \in \Sigma$, $(\gamma_1, \gamma_2) \in \Gamma_1 \times \Gamma_2$, $\delta((q_1, q_2), \sigma, (\gamma_1, \gamma_2)) = \delta_{\mathscr{A}_1}(q_1, \sigma, \gamma_1) \times \delta_{\mathscr{A}_2}(q_2, \sigma, \gamma_2)$;
- $F = F_{\mathscr{A}_1} \times F_{\mathscr{A}_2}$;
- $R = \{Z \times Q_{\mathscr{A}_2} \mid Z \in R_{\mathscr{A}_2}\} \cup \{Q_{\mathscr{A}_1} \times Z \mid Z \in R_{\mathscr{A}_2}\}$.

Consider an HLTL$^{\lightning}_{\text{L}}$ formula ϕ over the set of atomic propositions $\mathcal{P}$ and a CHA$^{\lightning}$ model $\mathscr{M} \in \text{CHA}^{\lightning}(2^{\mathcal{P}}, \Gamma, Q, B)$. Let $\mathscr{F}$ be the automata obtained from the negation of ϕ as described in Algorithm 2. As already noted, $\mathscr{M}$ and $\mathscr{F}$ need not to be box-compatible. In order to make the asynchronous product always computable, the *box alignment* procedure is introduced. If $\mathscr{M} \propto_b \mathscr{F}$, then the box alignment procedure leaves both automata unchanged. On the contrary, if $\mathscr{M} \not\propto_b \mathscr{F}$, then the box alignment procedure produces two box-compatible automata $\mathscr{M}'$ and $\mathscr{F}'$ by appropriately "padding", with suitably-defined children, the boxes violating condition (i) in Definition 32. $\mathscr{M}'$ and $\mathscr{F}'$ are defined in such

Algorithm 3 Box alignment procedure

signature BoxAlign $\mathrm{CSA}^{\natural} \times \mathrm{CSA}^{\natural} \to \mathrm{CSA}^{\natural} \times \mathrm{CSA}^{\natural}$
ensure The returned $\mathrm{CSA}^{\natural}$ are box-compatible
1: **function** BoxAlign($\mathscr{M}, \mathscr{F}$)
2: $(\mathscr{M}', \mathscr{F}') \leftarrow (\mathscr{M}, \mathscr{F})$
3: $m \leftarrow max\left\{\{|\beta_{\mathscr{M}}(b_m)| \mid b_m \in B_{\mathscr{M}}\} \cup \{|\beta_{\mathscr{F}}(b_f)| \mid b_f \in B_{\mathscr{F}}\}\right\}$
4: **for all** $b_m \in B_{\mathscr{M}'}$ **do**
5: **while** $|\beta_{\mathscr{M}'}(b_m)| < m$ **do**
6: $\beta_{\mathscr{M}'}(b_m) \leftarrow \beta_{\mathscr{M}'}(b_m) \cdot \mathscr{A}_{\mathsf{pad}}$
7: $\delta_{\mathscr{M}'} \leftarrow \mathsf{updateTransitionsRef}(b_m, \delta_{\mathscr{M}'})$
8: **end while**
9: **end for**
10: **for all** $b_f \in B_{\mathscr{F}'}$ **do**
11: **while** $|\beta_{\mathscr{F}'}(b_f)| < m$ **do**
12: $\beta_{\mathscr{F}'}(b_f) \leftarrow \beta_{\mathscr{F}'}(b_f) \cdot \mathscr{A}_{\mathsf{pad}}$
13: $\delta_{\mathscr{F}'} \leftarrow \mathsf{updateTransitionsCheck}(b_f, \delta_{\mathscr{F}'})$
14: **end while**
15: **end for**
16: **for all** $(b_m, b_f) \in B_{\mathscr{M}'} \times B_{\mathscr{F}'}$ **do**
17: $(\beta_{\mathscr{M}'}, \beta_{\mathscr{F}'}) \leftarrow (\varepsilon, \varepsilon)$
18: **for** $0 \leq i < |\beta_{\mathscr{M}'}(b_m)|$ **do**
19: $(\mathscr{A}_{\mathscr{M}}, \mathscr{A}_{\mathscr{F}}) \leftarrow$ BoxAlign$(\beta_{\mathscr{M}'}(b_m)_i, \beta_{\mathscr{F}'}(b_f)_i)$
20: $\beta_{\mathscr{M}'}(b_m) \leftarrow \beta_{\mathscr{M}'} \cdot \mathscr{A}_{\mathscr{M}}$
21: $\beta_{\mathscr{F}'}(b_f) \leftarrow \beta_{\mathscr{F}'} \cdot \mathscr{A}_{\mathscr{F}}$
22: **end for**
23: **end for**
24: **return** $(\mathscr{M}', \mathscr{F}')$
25: **end function**

a way that the model checking result is not altered. More precisely, the *box alignment* procedure is described in Algorithm 3 and proceeds as follows. In line 3, the maximum number of children required by any box in both systems is stored in the variable m. The loop in lines 4–9 iterates over every box b_m in the model automaton and, if a box is refined by less than m automata, an instance of the special padding automaton $\mathscr{A}_{\mathsf{pad}}$ is added until the box is refined by exactly m automata. The padding automaton $\mathscr{A}_{\mathsf{pad}}$, whose topology is shown in Figure 3.6, will be discussed later. At this point, it suffices to say that $\mathscr{A}_{\mathsf{pad}}$ is defined in such a way that no run entering its en_{ref} initial state can be accepting. After adding an instance of $\mathscr{A}_{\mathsf{pad}}$ to the refinements of the box b_m, it is necessary to update the transition relation $\delta_{\mathscr{M}'}$ to properly extend the list of entering states for transitions entering b_m. This is done by means of a function updateTransitionsRef taking as inputs the box b_m and the original transition relation $\delta_{\mathscr{M}'}$ and returning an updated transition

relation $\delta'_{\mathscr{M}'}$ defined as the union of the following sets of edges:

- $\{(s, \sigma, \gamma, t) \in \delta_{\mathscr{M}'} \mid t \neq (b_m, \overline{en})\}$, the set of original edges not entering b_m;
- $\{(s, \sigma, \gamma, t') \mid (s, \sigma, \gamma, t) \in \delta_{\mathscr{M}'},\ t = (b_m, \overline{en})\}$ and $t' = (b_m, \overline{en} \cdot en_{\text{ref}})$, the set of edges substituting the original ones entering b_m and specifying the additional en_{ref} entering state for the newly-added padding automaton.

The intuition behind this is to allow for the product to be defined while not introducing accepting computations in which the newly-introduced system automata are involved.

In lines 10–15, the procedure iterates over every box b_f in the formula automaton $\mathscr{F}'$ and, similarly to the previous loop, if a box is refined by less automata than m, padding automata are added until the box is refined by exactly m automata. When padding the formula automaton, it is necessary to distinguish between two cases, depending on whether the current run allows for the existence of additional children or not. If the box allows for the existence of an additional children, computations involving the newly-introduced padding automaton might be accepting. On the contrary, if the box explicitly forbids the existence of an additional children, no run involving the newly-introduced padding-automaton must be allowed to be accepting. This discrimination is taken care of partly by the function updateTransitionsCheck used to update the transition relation after the addition of a padding automaton and partly by the padding automaton itself, as explained later. Before addressing the details, notice that the existence of the i-th child of a box (α, d) can be explicitly forbidden if:

(i) the father explicitly requires the non-existence of the i-th child, i.e. $\neg\downarrow_i \top \in \alpha$;

(ii) the previous sibling's entering state in the current run is such that the corresponding atom contains $\neg \rightarrow \top$. Notice that the previous sibling automaton could have more than one entering state, and some of them may allow for the existence of a right sibling and some not. For a concrete example it suffices to think of formulae of the form $\downarrow_{i-1}(p \vee \neg \rightarrow \top)$ belonging to α;

(iii) the previous sibling, at some later point in time, requires for the non-existence of a right sibling. This happens for example when formulae of the form $\downarrow_{i-1}(\mathtt{X}(\neg \rightarrow \top))$ or $\downarrow_{i-2}(\rightarrow(p\,\mathtt{U}\,(\neg \rightarrow \top)))$ belong to α.

The transition update function updateTransitionsCheck takes care of enforcing the non-acceptance of runs involving the newly-added padding automaton in cases (i) and (ii). For the sake of clarity, before continuing with the formalization, the function $atom : Q_{\mathscr{F}} \mapsto \text{Atm}$ is defined as the function associating to each (possibly structured) state in the formula automaton the corresponding atom. $atom(q) = \emptyset$ if q is one of the states of the padding automaton $\mathscr{A}_{\text{pad}}$. The function updateTransitionsCheck takes as inputs the box b_f and the original transition relation $\delta_{\mathscr{F}'}$ and returns an updated transition relation $\delta'_{\mathscr{F}'}$ defined as the union of the following sets of edges:

- $\{(s, \sigma, \gamma, t) \in \delta_{\mathscr{F}'} \mid t \neq (b_f, \overline{en})\}$, the set of original edges not entering b_f, which are left unchanged;

- $\{(s, \sigma, \gamma, t') \mid (s, \sigma, \gamma, t) \in \delta_{\mathscr{F}'}, t = (b_f, \overline{en}), t' = \mathsf{getTrg}(t)\}$, with the function getTrg defined as follows :

$$\mathsf{getTrg}((b_f, \overline{en})) = \begin{cases} (b_f, \overline{en} \cdot en_{\mathrm{ref}}), & \text{if } \neg\downarrow_{|\overline{en}|+1} \top \in atom(b_f) \vee \neg \rightarrow \top \in atom(last(\overline{en})); \\ (b_f, \overline{en} \cdot en), & \text{otherwise.} \end{cases}$$

The original edges entering the padded formula box are substituted by new edges in such a way that computations involving the newly-added automaton falling in cases (i) (i.e. $\neg\downarrow_{|\overline{en}|+1} \top \in atom(b_m)$) and (ii) (i.e. $\neg \rightarrow \top \in atom(last(\overline{en}))$) are forced to enter the padding automaton by its en_{ref}, which as mentioned before, forces the run to be non-accepting. Unfortunately, at this point, it is not as easy to identify runs falling in case (iii). So, in every other case, the updateTransitionsCheck forces the runs to enter the padding automaton by its *en* state, which could lead to both non-accepting or accepting runs. The discrimination of runs falling in case (iii) is taken care of by the structure of the padding automaton itself, which is now described in detail.

Definition 34 (The padding automaton $\mathscr{A}_{\mathrm{pad}}$)**.** The padding automaton, depicted in Figure 3.6, is a CSA$^{\xi}$ defined on the same sets of synchronization symbols Γ and input alphabet Σ as his siblings as

$$\mathscr{A}_{\mathrm{pad}} = \langle Q_{\mathrm{pad}}, En_{\mathrm{pad}}, Ex_{\mathrm{pad}}, B_{\mathrm{pad}}, \beta_{\mathrm{pad}}, \delta_{\mathrm{pad}}, F_{\mathrm{pad}}, R_{\mathrm{pad}} \rangle,$$

where:

- $Q_{\mathrm{pad}} = \{en, en_{\mathrm{ref}}, L_{\mathrm{acc}}, L_{\mathrm{ref}}, B_{\mathrm{acc}}, B_{\mathrm{ref}}, ex\}$;
- $En_{\mathrm{pad}} = \{en, en_{\mathrm{ref}}\}$;
- $Ex_{\mathrm{pad}} = \{ex\}$;
- $B_{\mathrm{pad}} = \{B_{\mathrm{acc}}, B_{\mathrm{ref}}\}$;
- $\beta_{\mathrm{pad}} = \{(B_{\mathrm{acc}}, \mathscr{A}_{\mathrm{pad}}), (B_{\mathrm{ref}}, \mathscr{A}_{\mathrm{pad}})\}$;
- $F_{\mathrm{pad}} = \{en, L_{\mathrm{acc}}, B_{\mathrm{acc}}, ex\}$;
- $R_{\mathrm{pad}} = \{\{en, L_{\mathrm{acc}}, B_{\mathrm{acc}}, ex\}\}$;
- δ_{pad} is defined as the union of the following sets of edges:

$$InRef = \left\{ (s, \sigma, \langle \overline{\alpha}, \overline{\tau} \rangle, t) \;\middle|\; \begin{array}{l} \sigma \in \Sigma, \langle \overline{\alpha}, \overline{\tau} \rangle \in \Gamma, s \in \{en_{\mathrm{ref}}, L_{\mathrm{ref}}, (B_{\mathrm{ref}}, ex), B_{\mathrm{ref}}\}, \\ t \in \{L_{\mathrm{ref}}, (B_{\mathrm{ref}}, en)\} \end{array} \right\};$$

$$InAcc = \left\{ (s, \sigma, \langle \overline{\alpha}, \overline{\tau} \rangle, t) \;\middle|\; \begin{array}{l} \sigma \in \Sigma, \langle \overline{\alpha}, \overline{\tau} \rangle \in \Gamma. \neg \rightarrow \top \notin last(\overline{\tau}), \\ s \in \{en, L_{\mathrm{acc}}, (B_{\mathrm{acc}}, ex), B_{\mathrm{acc}}\}, \\ t \in \{L_{\mathrm{acc}}, (B_{\mathrm{acc}}, en), ex\} \end{array} \right\};$$

$$ToRef = \left\{ (s, \sigma, \langle \overline{\alpha}, \overline{\tau} \rangle, t) \;\middle|\; \begin{array}{l} \sigma \in \Sigma, \langle \overline{\alpha}, \overline{\tau} \rangle \in \Gamma. \neg \rightarrow \top \in last(\overline{\tau}), \\ s \in \{en, L_{\mathrm{acc}}, (B_{\mathrm{acc}}, ex), B_{\mathrm{acc}}\}, \\ t \in \{L_{\mathrm{ref}}, (B_{\mathrm{ref}}, en)\} \end{array} \right\}.$$

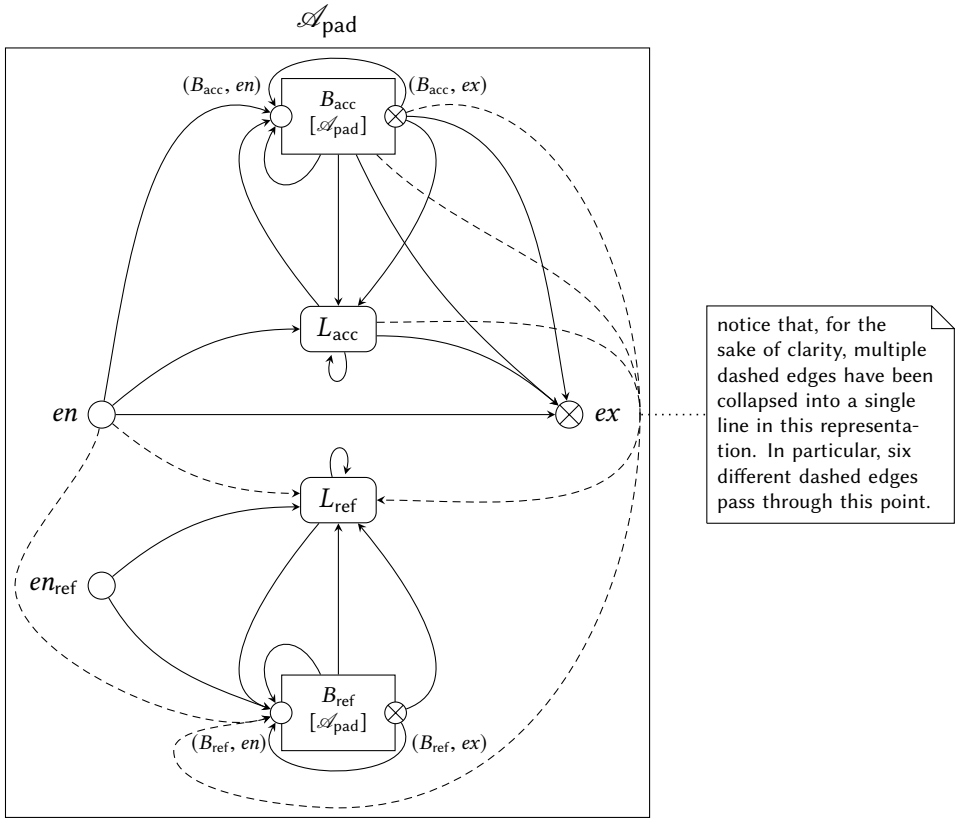


Figure 3.6: Topological structure for $\mathscr{A}_{\text{pad}}$

Intuitively, $\mathscr{A}_{\text{pad}}$ can be seen as partitioned in two regions: one "accepting" region, consisting in the states in F_{pad}, and one "non-accepting" region with states in $Q_{\text{pad}} \setminus F_{\text{pad}}$. As already mentioned, computations entering the non-accepting region are forced to remain in it, cannot reach the exiting state and thus cannot be accepting. The edges for the non-accepting region are indeed defined in set *InRef* in Definition 34 and link, for each input symbol and synchronization symbol, each refuting source in $\{en_{\text{ref}}, L_{\text{ref}}, (B_{\text{ref}}, ex), B_{\text{ref}}\}$ to each refuting target in $\{L_{\text{ref}}, (B_{\text{ref}}, en)\}$. While staying in the accepting region, the padding automaton must "monitor" the behaviour of the last "real" (i.e. not added by the padding procedure) sibling and, whenever such automaton enters a state whose atom explicitly requires $\neg \longrightarrow \top$, the padding automaton must perform a transition to its non-accepting region, ensuring that the current run cannot be accepted. This monitoring process is made possible by the second element in the synchronization symbol of each transition, which is equal to the target's atom, and allows for the definition of the remaining transitions of $\mathscr{A}_{\text{pad}}$ in sets *InAcc*, *ToRef* in Definition 34. In particular, *InAcc* contains all those edges in which the last "real" child transits in states allowing for the existence of a right sibling. In this cases, the transitions remain inside the accepting region. *ToRef* contains all those edges in which the last "real" child transits in states that explicitly forbids the existence of a right sibling. In those case $\mathscr{A}_{\text{pad}}$ transits to its non-accepting region and the computation

is rejected. In Figure 3.6, classes of transitions having the same source and target are represented by edges between states. For edges whose source is a state in F_{pad}, transitions belonging to *ToRef* are depicted by dashed lines, while the others in *InAcc* are represented by solid lines.

Getting back to the box alignment procedure, after the previously-discussed loops in lines 4–15, every box in both $\mathscr{M}'$ and $\mathscr{F}'$ is refined by exactly m machines, thus condition (i) in Definition 32 is satisfied. What remains is to ensure that also condition (ii) holds, which is done by means of the loops in lines 16–23, with recursive calls to the BoxAlign procedure (line 19).

Notice that, after applying the box alignment procedure, the CHA$^{\hbar}$ $\mathscr{M}$ and $\mathscr{F}$ may be transformed in recursive automata (CSA$^{\hbar}$), since instances of the recursive machine $\mathscr{A}_{\text{pad}}$ might have been be introduced. In particular, the results $\mathscr{M}'$ and $\mathscr{F}'$ of the box alignment procedure belong to the sub-class of *Simple* CSA$^{\hbar}$ (SCSA$^{\hbar}$), in which the only automaton allowed to perform recursive calls is $\mathscr{A}_{\text{pad}}$. The synchronous product of SCSA$^{\hbar}$, as per Definition 33, may be a recursive automaton in which the only recursive machines are products of $\mathscr{A}_{\text{pad}}$. In fact, the synchronous product of a CHA$^{\hbar}$ and $\mathscr{A}_{\text{pad}}$ is always a CHA$^{\hbar}$. More formally, the only recursive machines in the synchronous product of two SCSA$^{\hbar}$ belong to the set Ξ_{pad}, namely the least set such that: (i) $\mathscr{A}_{\text{pad}} \in \Xi_{\text{pad}}$; (ii) $\mathscr{A}, \mathscr{B} \in \Xi_{\text{pad}} \Rightarrow \mathscr{A} \otimes \mathscr{B} \in \Xi_{\text{pad}}$. Such machines are equivalent to a single $\mathscr{A}_{\text{pad}}$, therefore when computing the product of two SCSA$^{\hbar}$, they can be substituted with an $\mathscr{A}_{\text{pad}}$, in order to obtain yet another SCSA$^{\hbar}$. It is possible to extend the emptiness procedure devised for CHA$^{\hbar}$ in Algorithm 1 to SCSA$^{\hbar}$ as shown in Algorithm 4. The SCSA$^{\hbar}$ emptiness procedure is rather similar to the CHA$^{\hbar}$ one described in Section 3.3, with the only difference being the way in which the InOut set is recursively computed in the Unbox subroutine. In fact, applying the CHA$^{\hbar}$ emptiness procedure to SCSA$^{\hbar}$ without any precaution may lead to non-termination of the algorithm. When computing the InOut set for a box, the new procedure (see lines 7–13) distinguishes between two cases depending on whether the box is refined by some recursive automata or not. If the box was not "padded", then the InOut set is computed exactly as in the previous procedure (see line 8). On the contrary, if the box is refined by instances of $\mathscr{A}_{\text{pad}}$, the emptiness result is computed only considering the "original" siblings, i.e. the siblings which are not $\mathscr{A}_{\text{pad}}$. This partial emptiness result is stored in the temporary variable I' and is used to compute the actual emptiness result for the box by means of the function getInOutSCSA (see lines 11–12). This function grounds on the fact that it is trivial to compute the emptiness result for $\mathscr{A}_{\text{pad}}$ by hand, and in particular Emptiness$(\mathscr{A}_{\text{pad}}) = \{(en, ex), (en, \top)\}$. The function getInOutSCSA takes as inputs the partial emptiness result I', the number n of automata for which the partial emptiness result was computed, the number m of total automata refining the current box, and builds the complete emptiness result as follows:

$$\text{getInOutSCSA}(I', n, m) = \left\{ (\overline{s}, \overline{t}) \;\middle|\; \begin{array}{l} \overline{s} = \overline{s}' \cdot en^{m-n} \text{ and } \overline{t} = \overline{t}' \cdot \overline{t}_{\text{pad}},\ \overline{t}_{\text{pad}} \in \{ex, \top\}^{m-n} \\ \text{and } \left(\overline{s}', \overline{t}'\right) \in I' \end{array} \right\}$$

Intuitively, getInOutSCSA extends each component of each element $(\overline{s}', \overline{t}')$ in the partial

Algorithm 4 SCSA$^{\not\downarrow}$ emptiness

signature EMPTINESS : $(\text{SCSA}^{\not\downarrow})^+ \to_{\overline{\mathscr{A}}} \text{InOut}(\overline{\mathscr{A}})$
1: **function** EMPTINESS($\overline{\mathscr{A}}$)
2: **return** *reach*(*prod*(UNBOX($\overline{\mathscr{A}}$)));
3: **end function**

signature UNBOX $(\text{SCSA}^{\not\downarrow})^+ \to CFA^+$
1: **function** UNBOX($\overline{\mathscr{A}}$)
2: $\overline{\mathscr{A}'} \leftarrow \varepsilon$
3: **for** $0 \leq i < |\overline{\mathscr{A}}|$ **do**
4: $\mathscr{A}' \leftarrow \overline{\mathscr{A}}_i$
5: **for all** $b \in B_{\mathscr{A}'}$ **do**
6: $n \leftarrow \max\{i \mid \beta_{\mathscr{A}'}(b)_i \neq \mathscr{A}_{\text{pad}}\} + 1$ ▷ n is the num. of "original" children
7: **if** $n = 0$ **then** ▷ the box contains no recursive automata
8: $I \leftarrow$ EMPTINESS($\beta_{\mathscr{A}'}(b)$) ▷ just as with CHA$^{\not\downarrow}$
9: **else** ▷ the box was "padded" and contains recursive automata
10: $m \leftarrow |\beta_{\mathscr{A}'}(b)|$ ▷ m is the total number of children
11: $I' \leftarrow$ EMPTINESS($\beta_{\mathscr{A}'}(b)_0, \ldots, \beta_{\mathscr{A}'}(b)_{n-1}$)
12: $I \leftarrow \text{getInOutSCSA}(I', n, m)$
13: **end if**
14: $Q_{\mathscr{A}'} \leftarrow Q_{\mathscr{A}'} \cup (\{b\} \times \text{I})$ ▷ add *summary* states
15: $B_{\mathscr{A}'} \leftarrow B_{\mathscr{A}'} \setminus \{b\}$ ▷ remove b from the set of boxes
16: $\beta_{\mathscr{A}'} \leftarrow \delta_{\mathscr{A}'} \restriction B_{\mathscr{A}'}$ ▷ restrict the box refinement function
17: $F_{\mathscr{A}'} \leftarrow \text{acc}(F_{\mathscr{A}'}, b, I)$ ▷ suitably enrich accepting states
18: $R_{\mathscr{A}'} \leftarrow \{\text{acc}(X, b, I) \mid X \in R_{\mathscr{A}'}\}$ ▷ suitably enrich accepting states
19: $\delta_{\mathscr{A}'} \leftarrow \text{trn}(Q_{\mathscr{A}'}, \beta_{\mathscr{A}'}, \delta_{\mathscr{A}'}, b, I)$ ▷ update the transition function
20: **end for**
21: $\overline{\mathscr{A}'} \leftarrow \overline{\mathscr{A}'} \cdot \mathscr{A}'$
22: **end for**
23: **return** $\overline{\mathscr{A}'}$
24: **end function**

InOut set I' to account for the additional padding automata refining the box.

With the emptiness procedure for SCSA$^{\not\downarrow}$ in place, the model checking procedure described in Figure 3.5 and summarized as follows is finally complete. The model checking procedure takes as inputs a CHA$^{\not\downarrow}$ model $\mathscr{A}$ on the alphabet $\Sigma = 2^{\mathcal{P}}$ and an HLTL$^{\not\downarrow}_{\text{L}}$ property ϕ on the set of atomic propositions $\mathcal{P}$. By applying Algorithm 2, the model checker builds the automaton $\mathscr{A}_{\neg\phi}$ accepting only hierarchical words satisfying $\neg\phi$. After that, since $\mathscr{A}$ and $\mathscr{A}_{\neg\phi}$ may as well be not box-compatible, the model checker applies the previously-discussed box alignment procedure in order to obtain two box-compatible automata $\mathscr{A}'$ and $\mathscr{A}'_{\neg\phi}$. These two automata belong to the class of SCSA$^{\not\downarrow}$, since they may contain the

recursive $\mathscr{A}_{\mathsf{pad}}$ automaton. Then, the synchronous product $\mathscr{A}' \otimes \mathscr{A}'_{\neg\phi}$ is computed as described in Definition 33, thus obtaining the product SCSA$^{\lightning}$. What remains is to apply the emptiness decision procedure for SCSA$^{\lightning}$ described in Algorithm 4. If the emptiness procedure returns an empty set, then the property ϕ is satisfied by every computation accepted by $\mathscr{A}$. On the contrary, every returned InOut sequence witnesses the existence of a run being both accepted by $\mathscr{A}$ and not satisfying ϕ.

Conclusions

This thesis work faced multiple key-aspects in the broader domain of model-based systems verification, ranging from formal specification languages, with the definition of the *Dynamic State Machines* (DSTM) modelling language in Chapter 1, to model-based testing techniques, which as of today are considered *leading-edge* in industry, in Chapter 2, to logics and model checking in Chapter 3.

During this thesis work, crucial flaws in the automatable test case generation procedure for DSTM models devised during previous research [5, 6] were detected and addressed by defining a novel procedure, as described in Chapter 2. The new procedure was also implemented in the existing software tool for test case generation developed during the previous research projects. Furthermore, a theoretical foundation for DSTM model checking was laid down in Chapter 3, with the introduction of HLTL$^{\lightning}$, an extension of the well-known Linear-time Temporal Logic (LTL) designed to expressively predicate on linear properties of interrupting hierarchical computations. A concrete instantiation of the logic's semantics was given in terms of Communicating Structured Automata with Interrupts (CSA$^{\lightning}$), which are slightly simpler machines than DSTM, but nonetheless manage to maintain the main characteristics of concurrency, hierarchy and interrupts. After restricting the focus to the local fragment of this logic, which is denoted by HLTL$^{\lightning}_{\mathrm{L}}$, and to the sub-class of non-recursive CSA$^{\lightning}$, which are called Communicating Hierarchical Automata (CHA$^{\lightning}$), several algorithmic results were achieved: a decision procedure for the emptiness problem of CHA$^{\lightning}$ was described in Section 3.3; an algorithm to decide the satisfiability of an HLTL$^{\lightning}_{\mathrm{L}}$ formulae over the class of CHA$^{\lightning}$ computations was given in Section 3.4; finally, a model checking procedure for CHA$^{\lightning}$ models against HLTL$^{\lightning}_{\mathrm{L}}$ properties was achieved in Section 3.5.

The natural prosecution of this research work would be the definition of a model checking procedure for DSTM models and HLTL$^{\lightning}$ properties, and the extension of the existing software tool to implement model checking capabilities.

Appendix A
Translating DSTM models to PROMELA: a complete example

A.1 The *Counting* DSTM model

Consider the *Counting* DSTM specification detailed in figure 1.1 and in table 1.1, whose flattening is described throughout example 8. This section, in listing A.1, provides a complete PROMELA encoding for the model, obtained as described in 2.4.5 with the addition of the global variables and declarations required for test case generation (see 2.5).

Listing A.1 PROMELA specification for the *Counting* DSTM model

```
1: #define MAX_PROC 4; // maximum number of concurrent processes
2: // Global variables, channels, datatypes declarations
3: byte x;
4: // Mtype declarations for each machine's state name
5: mtype = {initial,idle1,counterBox,interrupted,stopped};
6: mtype = {default,idle2,wait,limit};
7: mtype = {byOne,byTwo,simpleIncr,doubleIncr,finished};
8: // Data objects needed to properly model the system
9: bit isFirstDescent = 1;
10: bit HasToken[MAX_PROC];
11: bit HasFired = 0;
12: bit dyingPid[MAX_PROC];
13: bit HasExecuted[MAX_PROC]; //set if pid executed in current step
14: bit descendantExecuted[MAX_PROC];
15: bit updateState = 0;
16: //structure needed to keep track of the process hierarchy
17: typedef childrenArray {
18:     bit children[MAX_PROC];
19: }
20: childrenArray ChildrenMatrix[MAX_PROC];
21: //global variables for test case gen.
22: mtype LastState, LastTransition;
23:
```

```
24: proctype Main(pid parent; mtype initial; chan chTerm;
25:   chan chTerm_ex) {
26:
27:   bit didBackProp = 0; byte i; pid pidTemp;
28:   //declare channels for termination synch. with children here
29:   chan chTerm_counterBox_Counter       = [1] of bit;
30:   chan chTerm_chTerm_counterBox_Counter_limit = [1] of bit;
31:
32:   mtype state=initial, nextState;
33:   LastState = initial;
34:
35:   do
36:   // State initial
37:   :: (state == initial && HasToken[_pid]) -> atomic {
38:     HasToken[_pid]=0;
39:     didBackProp=0;
40:     if
41:     // Transition T1
42:     :: ((1) && !descendantExecuted[_pid]) ->
43:         state = idle1; HasFired=1;
44:         HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
45:         LastState = state; LastTransition = T1;
46:     :: else -> //no transition is executable
47:       if
48:       :: (!HasExecuted[_pid]) ->
49:         // if this proc did not exec. in this step
50:         for (i : 0 .. MAX_PROC-1) { // pass token to children
51:           if
52:           ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
53:           ::else->skip;
54:           fi;
55:         }
56:       ::else->skip;
57:       fi;
58:     fi;
59:     nextState = state; state = backProp;
60:   }
61:   // State idle1
62:   :: (state == idle1 && HasToken[_pid]) -> atomic {
63:     HasToken[_pid]=0;
64:     didBackProp=0;
65:     if
66:     // Transition T2
67:     :: ((1) && !descendantExecuted[_pid]) ->
68:       state = counterBox; HasFired=1;
69:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
70:       LastState = state; LastTransition = T2;
71:       pidTemp = run Counter(_pid,default,
72:         chTerm_counterBox_Counter,
73:         chTerm_counterBox_Counter_limit, 100);
74:       ChildrenMatrix[_pid].children[pidTemp] = 1;
75:     :: else -> //no transition is executable
```

```
76:       if
77:       :: (!HasExecuted[_pid]) ->
78:         // if this proc did not exec. in this step
79:         for (i : 0 .. MAX_PROC-1) { // pass token to children
80:           if
81:           ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
82:           ::else->skip;
83:           fi;
84:         }
85:       ::else->skip;
86:       fi;
87:     fi;
88:     nextState = state; state = backProp;
89:   }
90:   // State counterBox
91:   :: (state == counterBox && HasToken[_pid]) -> atomic {
92:     HasToken[_pid]=0;
93:     didBackProp=0;
94:     if
95:     // Transition T3 (return by interrupt)
96:     :: ((signal?) && !descendantExecuted[_pid]) ->
97:       state = interrupted; HasFired=1;
98:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
99:       LastState = state; LastTransition = T3;
100:      chTerm_counterBox_Counter!<1>;
101:    // Transition T4 (return by interrupt)
102:    :: ( chTerm_counterBox_Counter_limit[?<1>]
103:      && !descendantExecuted[_pid]) ->
104:      state = stopped; HasFired=1;
105:      HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
106:      LastState = state; LastTransition = T4;
107:      chTerm_counterBox_Counter!<1>;
108:    :: else -> //no transition is executable
109:      if
110:      :: (!HasExecuted[_pid]) ->
111:        // if this proc did not exec. in this step
112:        for (i : 0 .. MAX_PROC-1) { // pass token to children
113:          if
114:          ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
115:          ::else->skip;
116:          fi;
117:        }
118:      ::else->skip;
119:      fi;
120:    fi;
121:    nextState = state; state = backProp;
122:  }
123:
124:  // handle upwards propagation of descendantExecuted
125:  ::(state==backProp && descendantExecuted[_pid]
126:    && !didBackProp)
127:    -> { didBackProp = 1; descendantExecuted[parent] = 1 }
```

```
128:   //handle original state restoring after backProp
129:   ::(state==backProp && updateState) ->
130:     { state = nextState; didBackProp=0 }
131:
132:   od unless (chTerm?[1] ||  dyingPid[parent]) -> {
133:     chTerm?1; dyingPid[_pid]=1
134:   }
135: }
136:
137: proctype Counter(pid parent; mtype initial; chan chTerm;
138:   chan chTerm_ex, int P_to) {
139:
140:   bit didBackProp = 0; byte i; pid pidTemp;
141:   //declare channels for termination synch. with children here
142:   chan chTerm_boxIncr1_Incrementer       = [1] of bit;
143:   chan chTerm_boxIncr2_Incrementer       = [1] of bit;
144:   chan chTerm_boxIncr1_Incrementer_finished = [1] of bit;
145:   chan chTerm_boxIncr2_Incrementer_finished = [1] of bit;
146:
147:   mtype state=initial, nextState;
148:   LastState = default;
149:
150:   do
151:   // state default
152:   :: (state == default && HasToken[_pid]) -> atomic {
153:     HasToken[_pid]=0;
154:     didBackProp=0;
155:     if
156:     // Transition T5
157:     :: ((1) && !descendantExecuted[_pid]) ->
158:       state = idle2; HasFired=1;
159:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
160:       LastState = state; LastTransition = T5;
161:     :: else -> //no transition is executable
162:       if
163:       :: (!HasExecuted[_pid]) ->
164:         // if this proc did not exec. in this step
165:         for (i : 0 .. MAX_PROC-1) { // pass token to children
166:           if
167:           ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
168:           ::else->skip;
169:           fi;
170:         }
171:       ::else->skip;
172:       fi;
173:     fi;
174:     nextState = state; state = backProp;
175:   }
176:   // state idle2
177:   :: (state == idle2 && HasToken[_pid]) -> atomic {
178:     HasToken[_pid]=0;
179:     didBackProp=0;
```

```
180:     if
181:     // Transition T6_T7_T8
182:     :: ((1) && !descendantExecuted[_pid]) ->
183:       state = wait; HasFired=1;
184:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
185:       LastState = state; LastTransition = T6_T7_T8;
186:       //run the two Incrementer machines
187:       pidTemp = run Incrementer(_pid,default,
188:         chTerm_boxIncr1_Incrementer,
189:         chTerm_boxIncr1_Incrementer_finished,P_to);
190:       ChildrenMatrix[_pid].children[pidTemp] = 1;
191:       pidTemp = run Incrementer(_pid,default,
192:         chTerm_boxIncr2_Incrementer,
193:         chTerm_boxIncr2_Incrementer_finished,P_to);
194:       ChildrenMatrix[_pid].children[pidTemp] = 1;
195:     :: else -> //no transition is executable
196:       if
197:       :: (!HasExecuted[_pid]) ->
198:         // if this proc did not exec. in this step
199:         for (i : 0 .. MAX_PROC-1) { // pass token to children
200:           if
201:           ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
202:           ::else->skip;
203:           fi;
204:         }
205:       ::else->skip;
206:       fi;
207:     fi;
208:     nextState = state; state = backProp;
209:   }
210:   // state wait
211:   :: (state == wait && HasToken[_pid]) -> atomic {
212:     HasToken[_pid]=0;
213:     didBackProp=0;
214:     if
215:     // Transition T9_T10_T11
216:     :: ((chTerm_boxIncr1_Incrementer_finished[?<1>])
217:       && !descendantExecuted[_pid]) ->
218:       state = limit; HasFired=1;
219:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
220:       LastState = limit; LastTransition = T9_T10_T11;
221:       //send termination message to machines
222:       chTerm_boxIncr1_Incrementer!<1>;
223:       chTerm_boxIncr2_Incrementer!<1>;
224:       chTerm_boxIncr1_Incrementer_finished?<_>;
225:       //reached exit state
226:       chTerm_ex!<1>;
227:     :: else -> //no transition is executable
228:       if
229:       :: (!HasExecuted[_pid]) ->
230:         // if this proc did not exec. in this step
231:         for (i : 0 .. MAX_PROC-1) { // pass token to children
```

```
232:           if
233:           ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
234:           ::else->skip;
235:           fi;
236:         }
237:       ::else->skip;
238:       fi;
239:     fi;
240:     nextState = state; state = backProp;
241:   }
242:
243:   // handle upwards propagation of descendantExecuted
244:   ::(state==backProp && descendantExecuted[_pid]
245:     && !didBackProp)
246:     -> { didBackProp = 1; descendantExecuted[parent] = 1 }
247:   //handle original state restoring after backProp
248:   ::(state==backProp && updateState) ->
249:     { state = nextState; didBackProp=0 }
250:
251:   od unless (chTerm?[1] ||  dyingPid[parent]) -> {
252:     chTerm?1; dyingPid[_pid]=1
253:   }
254: }
255:
256: proctype Incrementer(pid parent; mtype initial; chan chTerm;
257:   chan chTerm_ex, int P_limit) {
258:
259:   bit didBackProp = 0; byte i; pid pidTemp;
260:   //declare channels for termination synch. with children here
261:
262:   mtype state=initial, nextState;
263:   LastState = default;
264:
265:   do
266:   // state byOne
267:   :: (state == byOne && HasToken[_pid]) -> atomic {
268:     HasToken[_pid]=0;
269:     didBackProp=0;
270:     if
271:     // Transition T12
272:     :: ((1) && !descendantExecuted[_pid]) ->
273:       state = simpleIncr; HasFired=1;
274:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
275:       LastState = state; LastTransition = T12;
276:     :: else -> //no transition is executable
277:       if
278:       :: (!HasExecuted[_pid]) ->
279:         // if this proc did not exec. in this step
280:         for (i : 0 .. MAX_PROC-1) { // pass token to children
281:           if
282:           ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
283:           ::else->skip;
```

```
284:           fi;
285:         }
286:       ::else->skip;
287:       fi;
288:     fi;
289:     nextState = state; state = backProp;
290:   }
291:   // state byTwo
292:   :: (state == byOne && HasToken[_pid]) -> atomic {
293:     HasToken[_pid]=0;
294:     didBackProp=0;
295:     if
296:     // Transition T13
297:     :: ((1) && !descendantExecuted[_pid]) ->
298:       state = doubleIncr; HasFired=1;
299:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
300:       LastState = state; LastTransition = T13;
301:     :: else -> //no transition is executable
302:       if
303:       :: (!HasExecuted[_pid]) ->
304:         // if this proc did not exec. in this step
305:         for (i : 0 .. MAX_PROC-1) { // pass token to children
306:           if
307:           ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
308:           ::else->skip;
309:           fi;
310:         }
311:       ::else->skip;
312:       fi;
313:     fi;
314:     nextState = state; state = backProp;
315:   }
316:   // state simpleIncr
317:   :: (state == simpleIncr && HasToken[_pid]) -> atomic {
318:     HasToken[_pid]=0;
319:     didBackProp=0;
320:     if
321:     // Transition T14
322:     :: ((x<P_limit) && !descendantExecuted[_pid]) ->
323:       state = simpleIncr; HasFired=1;
324:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
325:       LastState = state; LastTransition = T14;
326:       x++; //action
327:     // Transition T16
328:     :: ((x<P_limit) && !descendantExecuted[_pid]) ->
329:       state = finished; HasFired=1;
330:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
331:       LastState = state; LastTransition = T16;
332:       x++; //action
333:       chTerm_ex!<1>; //send termination signal
334:     :: else -> //no transition is executable
335:       if
```

```
336:       :: (!HasExecuted[_pid]) ->
337:         // if this proc did not exec. in this step
338:         for (i : 0 .. MAX_PROC-1) { // pass token to children
339:           if
340:           ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
341:           ::else->skip;
342:           fi;
343:         }
344:       ::else->skip;
345:       fi;
346:     fi;
347:     nextState = state; state = backProp;
348:   }
349:   // state doubleIncr
350:   :: (state == doubleIncr && HasToken[_pid]) -> atomic {
351:     HasToken[_pid]=0;
352:     didBackProp=0;
353:     if
354:     // Transition T15
355:     :: ((1) && !descendantExecuted[_pid]) ->
356:       state = doubleIncr; HasFired=1;
357:       HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
358:       LastState = state; LastTransition = T15;
359:       x=x+2; //action
360:     :: else -> //no transition is executable
361:       if
362:       :: (!HasExecuted[_pid]) ->
363:         // if this proc did not exec. in this step
364:         for (i : 0 .. MAX_PROC-1) { // pass token to children
365:           if
366:           ::(ChildrenMatrix[_pid].children[i])->HasToken[i]=1;
367:           ::else->skip;
368:           fi;
369:         }
370:       ::else->skip;
371:       fi;
372:     fi;
373:     nextState = state; state = backProp;
374:   }
375:   // handle upwards propagation of descendantExecuted
376:   ::(state==backProp && descendantExecuted[_pid]
377:     && !didBackProp)
378:     -> { didBackProp = 1; descendantExecuted[parent] = 1 }
379:   //handle original state restoring after backProp
380:   ::(state==backProp && updateState) ->
381:     { state = nextState; didBackProp=0 }
382:
383:   od unless (chTerm?[1] ||  dyingPid[parent]) -> {
384:     chTerm?1; dyingPid[_pid]=1
385:   }
386: }
387:
```

```
388: active proctype Engine() {
389:   pid PidMain; byte i;
390:   chan chTerm_Main = [1] of {bit};
391:   chan chTerm_Main_exit = [1] of {bit};
392:   PidMain = run Main(_pid,initial,chTerm_Main,chTerm_Main_exit);
393:   ChildrenMatrix[_pid].children[PidMain]=1;
394:
395:   nextStep: // starts a new step
396:     atomic {
397:       // handle external channels management
398:       updateState=0
399:       HasFired=0;
400:       isFirstDescent=1;
401:       for (i : 0 .. MAX_PROC-1){
402:         HasExecuted[i]=0;
403:         descendantExecuted[i]=0;
404:         HasToken[i] = ChildrenMatrix[_pid].children[i];
405:       }
406:     }
407:     goto waitTimeout;
408:
409:   nextPhase: // starts a new phase in the current step
410:     atomic {
411:       updateState=0;
412:       HasFired=0;
413:       for ( i : 0 .. MAX_PROC - 1){
414:         // give token to engine's children
415:         HasToken[i] = ChildrenMatrix[_pid].children[i];
416:       }
417:       isFirstDescent = 0; //It's at least the second one
418:     }
419:     goto waitTimeout;
420:
421:   waitTimeout:
422:     do
423:     :: timeout -> //deadlock
424:       if
425:       :: (!HasFired && isFirstDescent) -> goto abort;
426:       :: (!HasFired && !isFirstDescent && !updateState) ->
427:         updateState = 1;
428:       :: (!HasFired && !isFirstDescent && updateState) ->
429:         goto nextStep;
430:       :: (HasFired && !updateState) -> updateState = 1;
431:       :: (HasFired && updateState) -> goto nextPhase;
432:       fi;
433:     od;
434:
435:   abort:
436:     dyingPid[_pid]=1;
437: }
```

Bibliography

[1] Edsger W. Dijkstra. “The humble programmer”. In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

[2] Leslie Lamport and Susan Owicki. “Proving liveness properties of concurrent programs”. In: *ACM Transactions on Programming Languages and Systems* 4 (1982).

[3] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[4] Horst Pflügl, Christian El-Salloum, and Ingrid Kundner. “Crystal Critical Systems Engineering Acceleration”. In: *ARTEMIS Magazine* 14 (June 2013), pp. 12–15.

[5] Massimo Benerecetti et al. “Dynamic state machines for modelling railway control systems”. In: *Science of Computer Programming* 133 (2017), pp. 116–153.

[6] Roberto Nardone et al. “Modeling railway control systems in Promela”. In: *International Workshop on Formal Techniques for Safety-Critical Systems*. Springer. 2015, pp. 121–136.

[7] Gregorio Barberio et al. “An interoperable testing environment for ERTMS-ETCS control systems”. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2014, pp. 147–156.

[8] David Lee and Mihalis Yannakakis. “Principles and methods of testing finite state machines-a survey”. In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123.

[9] David Harel. “Statecharts: A visual formalism for complex systems”. In: *Science of computer programming* 8.3 (1987), pp. 231–274.

[10] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. OMG Document Number formal/17-12-05 (`https://www.omg.org/spec/UML/2.5.1`).

[11] David Harel and Amnon Naamad. “The STATEMATE semantics of statecharts”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5.4 (1996), pp. 293–333.

[12] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. “Communicating Hierarchical State Machines”. In: *Automata, Languages and Programming*. Ed. by Jiri Wiedermann and Mogens van Emde Boas Peterand Nielsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 169–178. ISBN: 978-3-540-48523-0.

[13] Rajeev Alur et al. "Analysis of Recursive State Machines". In: *ACM Trans. Program. Lang. Syst.* 27.4 (July 2005), pp. 786–818. ISSN: 0164-0925. DOI: 10.1145/1075382.1075387. URL: http://doi.acm.org/10.1145/1075382.1075387.

[14] Edmund M Clarke and Wolfgang Heinle. *Modular translation of Statecharts to SMV*. Tech. rep. Citeseer, 2000.

[15] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. "Implementing statecharts in PROMELA/SPIN". In: *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*. IEEE. 1998, pp. 90–101.

[16] Toni Jussila et al. "Model checking dynamic and hierarchical UML state machines". In: *Proc. MoDeV2a: Model Development, Validation and Verification* (2006), pp. 94–110.

[17] *SPIN- Formal Verification*. URL: http://spinroot.com/ (visited on 10/18/2018).

[18] Gerard J. Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. First. Addison-Wesley Professional, 2003. ISBN: 0-321-22862-6.

[19] Gerard J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295.

[20] Moshe Y. Vardi and Pierre Wolper. "An automata-theoretic approach to automatic program verification". In: *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society. 1986, pp. 322–331.

[21] *PROMELA reference – never*. URL: http://spinroot.com/spin/Man/never.html (visited on 10/22/2018).

[22] Rajeev Alur and Mihalis Yannakakis. "Model Checking of Hierarchical State Machines". In: *SIGSOFT Softw. Eng. Notes* 23.6 (Nov. 1998), pp. 175–188. ISSN: 0163-5948. DOI: 10.1145/291252.288305. URL: http://doi.acm.org/10.1145/291252.288305.

[23] A. Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[24] Massimo Benerecetti, Ruggero Lanotte, Fabio Mogavero, and Adriano Peron. "Reasoning about Hierarchical Concurrent Computations". Unpublished.