Mapping UML-RT State Machines to kiltera

Technical Report 2010-569

Ernesto Posse

Applied Formal Methods Group
School of Computing
Queen's University
Kingston, Ontario, Canada

April 15, 2010

Contents

1 Introduction

The goal of this report is to formally specify a mapping from UML-RT State Machines into a process algebra called kiltera. More concretely, we define a mapping $\mathcal{T}[\![\cdot]\!] : \mathsf{SM} \rightarrow \mathsf{KLT}$ from the set SM of all syntactically valid State Machines to the set KLT of kiltera process terms. This is, for every State Machine $s \in \mathsf{SM}$ we want to define a corresponding kiltera term $\mathcal{T}[\![s]\!] \in \mathsf{KLT}$.

In order to specify the mapping of State Machines onto kiltera processes we use a textual syntax for State Machines, presented in Subsection 3.2. We also use a subset of the kiltera language, called the π_{klt} calculus, whose syntax is described in Subsection 2.2.1.

This report is organized as follows: In Section 2 we present background on UML-RT and kiltera. In Section 3 we introduce a textual syntax for UML-RT State Machines. In Section 4 we develop the map itself. Finally 5 concludes.

Rather than presenting the map in one go, we proceed gradually. First, we show how the hierarchical structure of State Machines is translated into kiltera in Subsection 4.1. In Subsection 4.2 we add basic (i.e., non-group) "sibling" transitions to the mapping. In Subsection 4.3 we introduce entry points and incoming transitions. In Subsection 4.4 we present exit points and outgoing transitions. In Subsection 4.5 we extend the mapping to support group transitions. In Subsection 4.6 we introduce a protocol that imposes priorities on conflicting transitions. We enhance the mapping with history states in Subsection 4.7. Then, in Subsection 4.8 we introduce actions abstractly and modify the mapping to encode the proper order of execution of entry, exit and transition actions.

2 Background

2.1 UML-RT

UML-RT is a dialect of the UML modeling language [3] used for specifying embedded and real time software systems. The language resulted from the combination of the Real-Time Object Oriented Modeling (ROOM) language [8] and general-purpose UML [7]. Tools supporting UML-RT include IBM Rational Rose Technical Developer toolkit [1], which is to be replaced by IBM Rational Software Architect Real Time Edition (IBM RSA-RTE) [2].

2.1.1 Structure diagrams

UML-RT allows the modeling of the system structure. through a hierarchy of capsules that are connected through typed ports. A capsule, as its name suggests, is a highly encapsulated entity, which communicates with other capsules only by sending and receiving signals through its ports. Therefore, a set of external ports owned by a capsule defines its interface. Each port has a type specified with a protocol, which identifies signals sent or received via the port. Capsules are organized hierarchically and each capsule may contain a number of instances of other capsules, called parts. External ports of these parts are connected (wired) statically or can be connected at run-time. Connected ports must implement the same protocol and be "compatible", i.e., the send signals of one port must be the receive signals of the other port and vice versa (in this case, one of the ports is said to be the base port and the other the conjugate port).

2.1.2 State Machines in UML-RT

The behaviour of a capsule is specified using UML-RT State Machines [7] which are similar to UML State Machines [3]. A UML-RT State Machine has hierarchical states and guarded transitions, which are triggered by signals received on ports. Each state declares its entry and exit actions and transitions have effects, so they can contain actions that are to be executed when the transition is fired. However, there also are some important syntactic and semantic differences between UML-RT State Machines and UML State Machines:

1) UML-RT State Machines cannot contain "and-states" (orthogonal regions). All states are "or-states". So, during execution a given UML-RT State Machine can be only in at most one simple state.

2) Transitions in UML-RT State Machines are not allowed to cross state boundaries and they may have explicit *entry* and *exit* points (here collectively called *connection points*). Hence, to represent a boundary-crossing transition, it must be broken up into segments, where each segment links connection points, either at the same level of nesting, or between a state and an immediate sub-state. During execution, connected segments build *a transition chain*, which is executed as one step.

3) In UML-RT entry points are by default connected to deep history pseudo-states. Suppose a composite state n is the target of a transition and that the associated entry point is *not* linked to a sub-state of n. If n has been visited previously, then the last sub-state visited in n is entered. This policy is applied recursively. Hence, entering a state can be interpreted as "resuming computation where it previously left off". In standard UML State Machines on the other hand, it is possible not to connect entry points to deep history pseudo-states, in which case an initial state is always entered, if an entry point is not explicitly connected to a sub-state. Just like in standard UML State Machines, event handling in UML-RT State Machines will follow a "run-to-completion" semantics: a state machine will handle one and only one event at a time, any transition chain enabled will be fully followed and its actions fully executed before the next event is handled.

UML-RT supports timing requirements using a special timing protocol and internal ports which implement this protocol. A capsule, which contains a port that implements the timing protocol, can schedule an event by sending a signal through this port. Scheduling can be a part of the entry or exit behavior of a state or as an action on a transition. After a specified amount of time, the capsule will receive a timeout event from the port which it can process as any other signal.

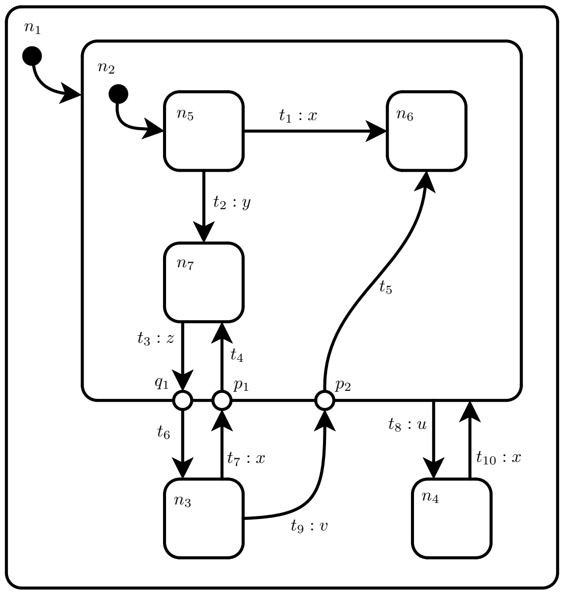

Figure 1: A simple UML-RT State Machine.

Example 1.

Consider the State Machine in Figure 1.

Transitions are marked as $t_i : x$ where t_i is the name of the transition, and x is its *trigger event*. State n_2 has two named entry points p_1 and p_2, and a named exit point q_1. There are several composite transition chains: $\langle t_3, t_6 \rangle$, $\langle t_7, t_4 \rangle$, $\langle t_9, t_5 \rangle$. Transition t_8 is a *group transition*, since its source is a composite state rather than a basic state. If transition t_{10} is taken, the last active sub-state of n_2 will be entered. Initially, the machine enters n_1, n_2 and n_5 in that order. If for example, the sequence of events $\langle y, u, x, z, v \rangle$ arrives, execution will proceed as follows. On y, take t_2 to n_7. On u, exit n_2 and take t_8 to n_4. On x, take t_{10}, enter n_2, and then enter n_7 (the last active sub-state of n_2). On z, take t_3 to q_1, exit n_2, take t_6 to n_3. Finally, on v, take t_9 to p_2, entering n_2, and then taking t_5 to n_6.

2.2 kiltera

kiltera is a language for modelling and simulating concurrent, interacting, real-time processes with support for mobility and distributed systems.

It is directly based on the π_{klt} calculus which is a real-time extension of the π calculus. The semantics of π_{klt} is given in terms of a Plotkin-style structural operational semantics over timed-labelled transition systems. The meta theory of π_{klt} extends that of the π calculus by a notion of time-bounded equivalence and a notion of timed compositionality and an associated timed congruence which allow reasoning about timed processes. The implementation of kiltera's interpreter is based on an abstract machine which has been proven sound with respect to π_{klt}'s operational semantics and uses event event scheduling as known in discrete-event simulation [10]. The interpreter supports two modes: real-time and simulated time. In real-time mode, the wall-clock timing of events reflects delays and timeouts specified in the model, and thus the interpreter actually pauses during idle periods. In simulated time, execution proceeds according to a logical clock, and events are processed as soon as they are available, thus avoiding idling when the model specifies events far apart in time. Consequently, execution in simulated time mode is more efficient, while execution in real-time mode is more reflective of the timing constraints (note that the interpreter is a prototype implemented in Python and does not use a real-time operating system; thus, even in real-time mode, timing constraints are only approximated).

Just like the π calculus, kiltera supports channel mobility. Furthermore, by assigning different kiltera processes to different machines, distributed simulation using the Time-Warp algorithm [4] is also supported. kiltera has been used for teaching in graduate courses at McGill and Queen's and the modelling of complex systems such as automobile traffic simulation. kiltera is available for download at `www.kiltera.org`.

Based on our experience so far, kiltera matches the features of UML-RT quite well. Moreover, kiltera's interpreter offers effective analysis.

2.2.1 π_{klt} syntax

To formally define the mapping we use the core of kiltera, the π_{klt} calculus, which has a mathematical notation suitable to describe the mapping.

Definition 1. (Syntax) The set KLT of π_{klt} **terms** is defined by the BNF in Table 1. Here P, P_i range over *process terms*, $x, y, ...$ range over the set of **(channel/event or variable) names**, A ranges over the set of **process names**, E ranges over *expressions*, and F ranges over *patterns*. n ranges over floating point numbers, s ranges over strings, and f ranges over function names, and the index set I is a subset $\{1, ..., n\} \subseteq \mathbb{N}$.

2.2.2 Informal Semantics

We now describe informally the semantics of the π_{klt}-calculus. For a formal semantics of the language see [5, 6].

- Expressions E are either constants ($\varnothing$ represents the *null* constant), variables (x), tuples of the form $\langle E_1, ..., E_m \rangle$ or function applications $f(E_1, ..., E_m)$. Patterns F have the same syntax as expressions, except that they do not include function applications.
- The process $\surd$ simply terminates.
- The process $x!E$ is a *trigger*; it triggers an event x with the value of E. Alternatively, we can say that it sends the value of E over a channel x. The expression E is optional: $x!$ is shorthand for $x!\varnothing$.

P	$::=$	$\surd$	Null
	$\mid$	$x!E$	Trigger/Output
	$\mid$	$\sum_{i\in I} x_i?F_i@y_i.P_i$	Listener/Input
	$\mid$	$\mathsf{new}\ x\ \mathsf{in}\ P$	New/Hide
	$\mid$	$\mathsf{wait}E.P$	Delay/Timer
	$\mid$	$\mathsf{if}\ E\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$	Conditionals
	$\mid$	$P_1 \parallel P_2$	Parallel
	$\mid$	$P_1; P_2$	Sequential composition
	$\mid$	$\mathsf{def}\ \{D_1; ...; D_n\}\ \mathsf{in}\ P$	Local definitions
	$\mid$	$A(x_1, ..., x_n)$	Instantiation/Call
D	$::=$	$\mathsf{proc}\ A(x_1, ..., x_n) = P$	Process definition
	$\mid$	$\mathsf{func}\ f(x_1, ..., x_n) = E$	Function definition
E	$::=$	$\varnothing \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \text{“}s\text{''} \mid x$	
	$\mid$	$\langle E_1, ..., E_m\rangle \mid f(E_1, ..., E_m)$	
F	$::=$	$\varnothing \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \text{“}s\text{''} \mid x$	
	$\mid$	$\langle F_1, ..., F_m\rangle$	

Table 1: π_{klt} syntax

- A process of the form $\sum_{i\in I} x_i?F_i@y_i.P_i$ is a *listener*. This process listens to all channels (or events) x_i, and when x_i is triggered with a value v that matches the pattern F_i, the corresponding process P_i is executed with y_i bound to the amount of time the listener waited, and the alternatives are discarded[1]. The suffixes F_i and $@y_i$ are optional: $x?.P$ is equivalent to $x?y@z.P$ for some fresh names y and z.

- The process $\mathsf{new}\ x\ \mathsf{in}\ P$ hides the name x from the environment, so that it is private to P. Alternatively, $\mathsf{new}\ \ x\ \mathsf{in}\ \ P$ can be seen as the creation of a new name, *i.e.*, a new event or channel, whose scope is P. We write $\mathsf{new}\ x_1, ..., x_n\ \mathsf{in}\ P$ for the process term $\mathsf{new}\ x_1\ \mathsf{in}\ \ ...\ \mathsf{new}\ \ x_n\ \mathsf{in}\ \ P$.

- The process $\mathsf{wait}\ E.P$ is a *delay*: it delays the execution of process P by an amount of time equal to the value of the expression E.[2]

[1]Note that to enable an input guard it is not enough for the channel to be triggered: the message must match the guard's pattern as well. Pattern-matching of inputs means that the input value must have the same "shape" as the pattern, and if successful, the free names in the pattern are bound to the corresponding values of the input. For example, the value $\langle 3, \mathsf{true}, 7\rangle$ matches the pattern $\langle 3, x, y\rangle$ with the resulting binding $\{{}^{\mathsf{true}}/_{x}, {}^{7}/_{y}\}$. The scope of these bindings is the corresponding P_i.

[2]The value of E is expected to be a non-negative real number. If the value of E is negative, $\Delta E.P$ cannot perform any action. Similarly, terms with undefined values (*e.g.*, $\Delta(1/0).P$) or with incorrectly typed expressions (*e.g.*, $\Delta\mathsf{true}.P$) cause the process to stop. Since the

- The process $\mathsf{if}\, E\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2$ is a conditional with the standard meaning.
- The process $P_1 \parallel P_2$ is the parallel composition of P_1 and P_2.
- The term $P_1; P_2$ is the sequential composition of P_1 and P_2.
- The term $\mathsf{def}\ \{D_1; ...; D_n\}\ \mathsf{in}\ \ P$ declares definitions D_i and executes P. The scope of these definitions is the entire term (so they can be invoked in P and in other definitions).
- The process $A(y_1, ..., y_n)$ creates a new instance of a process defined by $\mathsf{proc}\ A(x_1, ..., x_n) = P$, defined in some enclosing scope, where the ports $x_1, ..., x_n$ are substituted in the body P by the channels (or values) $y_1, ..., y_n$.

3 A syntax for UML-RT State Machines

3.1 Sequences

In the sequel we use several operations on sequences. In this Subsection we define the notation for these operations.

Notation 1. We write $1..k$ for the set $\{1, 2, ..., k\}$. Sequences will be enclosed in $\langle$ and $\rangle$. A sequence name will be denoted with an arrow on top, and its elements subscripted with their index, beginning from 1: $\vec{x} = \langle x_1, x_2, x_3, ...\rangle$. A finite sequence $\langle a_1, ..., a_k\rangle$ will be abbreviated as $a_{1..k}$. The empty sequence is denoted $\langle\rangle$, or ϵ.

Sequence concatenation will be denoted $\cdot$, so

$$\langle a_1, ..., a_k\rangle \cdot \langle b_1, ..., b_l\rangle \overset{def}{=} \langle a_1, ..., a_k, b_1, ..., b_k\rangle$$

3.2 State Machine syntax

We use a mathematical notation for State Machines, adapted from [9], which allows us to define the mapping compositionally.

In the sequel we will use the following sets:

- $\mathcal{N}_S$: the set of all possible *state names; we use $n, n_1, n_2, ..., m, ...$ for elements in $\mathcal{N}_S$*;
- $\mathcal{N}_A$: the set of all possible *entry point names; we use $p, p_1, p_2, ...$ for elements in $\mathcal{N}_A$*;
- $\mathcal{N}_B$: the set of all possible *exit point names; we use $q, q_1, q_2, ...$ for elements in $\mathcal{N}_B$;*
- $\mathcal{N}_C \overset{def}{=} \mathcal{N}_A \cup \mathcal{N}_B$: the set of all *connection point names*;

language is untyped we do not enforce these constraints statically.

- $\mathcal{E}$: the set of all possible *trigger events*; we use $x, x_1, x_2, ..., y, y_1, y_2, ..., z, z_1, z_2, ...$ for elements in $\mathcal{E}$;
- $\mathcal{E}_\perp \overset{def}{=} \mathcal{E} \cup \{\perp\}$: the set of events including the "non-event" $\perp$, used to mark transitions without a trigger;
- $\mathcal{A}$: the set of all possible *actions*; we use $a, a_1, a_2, ...$ for transition actions, en for entry actions and ex for exit actions in $\mathcal{A}$;
- $\mathcal{A}_\perp \overset{def}{=} \mathcal{A} \cup \{\perp\}$: the set of actions including the "non-action" $\perp$, i.e. the action that does nothing;
- $\mathbb{B} \overset{def}{=} \{\mathsf{false}, \mathsf{true}\}$ the set of boolean values;
- $\mathbb{N}$: the set of natural numbers

Furthermore, we make the following assumptions about these sets:

- Every state and connection point is labelled with a unique name[3];
- For every state name $n \in \mathcal{N}_S$, there is an entry point name $\mathsf{den}_n \in \mathcal{N}_A$ and an exit point name $\mathsf{dex}_n \in \mathcal{N}_B$. These denote the default entry and exit points of a state respectively, this is, when state n is the target of a transition, but the transition is not connected to any named entry point, it is assumed to be connected to the default entry point den_n. Analogously, when n is the source of a transition, and the transition doesn't leave the state from a named exit point, it is assumed to begin at the default exit point dex_n.

Before we define State Machine terms, we define the encoding of transitions, which link connection points. We distinguish between three kinds of transition: *incoming*, *outgoing* and *sibling*. Incoming transitions are transitions from an entry point to some sub-state. Outgoing transitions are transitions from a sub-state to an exit point. Sibling transitions are transitions between sub-states.

Definition 2. (Transitions) Let $\mathcal{K} = \{\mathsf{in}, \mathsf{out}, \mathsf{sib}\}$ represent the set of transition ***kinds***, (respectively in for incoming, out for outgoing, and sib for sibling). The set of all possible transitions is $\mathsf{TR} \overset{def}{=} \mathcal{K} \times \mathbb{B} \times \mathcal{N}_C \times \mathcal{N}_C \times \mathcal{E}_\perp \times \mathcal{A}_\perp$. Given

[3]If this is not the case, a simple traversal of the State Machine can give unique names, for example by providing fully qualified names or attaching a unique id.

a transition $t = (k, f, o, d, e, a) \in \mathsf{TR}$ we define the following functions:[4]

$$\begin{array}{rll} \mathsf{kind}(t) \overset{def}{=} k & & \text{The kind of transition} \\ \mathsf{firstinchain}(t) \overset{def}{=} f & & \text{Whether } t\text{is the first in a chain} \\ \mathsf{src}(t) \overset{def}{=} o & & \text{The source of the transition} \\ \mathsf{trg}(t) \overset{def}{=} d & & \text{The target of the transition} \\ \mathsf{evt}(t) \overset{def}{=} e & & \text{The trigger event of the transition} \\ \mathsf{act}(t) \overset{def}{=} a & & \text{The action of the transition} \end{array}$$

Now we can define State Machine terms.

Definition 3. (State Machine terms) The set SM of State Machine terms is defined according to the following BNF:

$$\begin{array}{rcll} s & ::= & [n, A, B, en, ex] & \text{Basic-state} \\ & \mid & [n, A, B, S, d, T, en, ex] & \text{Composite state} \end{array}$$

Here $n \in \mathcal{N}_S$ is the name of a state, $A \subseteq \mathcal{N}_A$ and $B \subseteq \mathcal{N}_B$ are the sets of entry and exit points where $A \cap B = \emptyset$ and $\mathsf{den}_n \in A$ and $\mathsf{dex}_n \in B$, $en, ex \in \mathcal{A}_\perp$ are the entry and exit actions, S is a sequence $\langle s_1, ..., s_k \rangle$ of sub-states with each $s_i \in \mathsf{SM}$, d is the index, in the sequence, of the default sub-state s_d, and $T \subseteq \mathsf{TR}$ is a set of transitions subject to the conditions stated below.

We first define the following useful functions for a given basic state $s = [n, A, B, ex, en]$:

$$\begin{array}{rll} \mathsf{name}(s) \overset{def}{=} n & & \text{The name of the state} \\ \mathsf{entries}(s) \overset{def}{=} A & & \text{The set of entry points of the state} \\ \mathsf{exits}(s) \overset{def}{=} B & & \text{The set of exit points of the state} \\ \mathsf{enact}(s) \overset{def}{=} en & & \text{The set of entry actions of the state} \\ \mathsf{exact}(s) \overset{def}{=} ex & & \text{The set of exit actions of the state} \end{array}$$

Analogously, for a composite state $s = [n, A, B, S, d, T, en, ex]$ with $S = s_{1..k}$,

[4]Note that since we assume unique names for all connection points, the source and target of a transition are well-defined.

we define

$$\begin{array}{rll}
\mathsf{name}(s) \overset{def}{=} n & & \text{The name of the state} \\
\mathsf{entries}(s) \overset{def}{=} A & & \text{The set of entry points of the state} \\
\mathsf{exits}(s) \overset{def}{=} B & & \text{The set of exit points of the state} \\
\mathsf{substates}(s) \overset{def}{=} S & & \text{The set of substates of the state} \\
\mathsf{transitions}(s) \overset{def}{=} T & & \text{The set of transitions of the state} \\
\mathsf{default}(s) \overset{def}{=} s_d & & \text{The default (initial) substate of the state} \\
\mathsf{enact}(s) \overset{def}{=} en & & \text{The set of entry actions of the state} \\
\mathsf{exact}(s) \overset{def}{=} ex & & \text{The set of exit actions of the state}
\end{array}$$

and all transitions $t \in T$ must satisfy the following conditions:

1. If $\mathsf{firstinchain}(t) = \mathsf{false}$ then $\mathsf{evt}(t) = \perp$
2. $\mathsf{kind}(t) = \mathsf{sib}$ if and only if there are sub-states s_i and s_j in S such that $\mathsf{src}(t) \in \mathsf{exits}(s_i)$ and $\mathsf{trg}(t) \in \mathsf{entries}(s_j)$.
3. $\mathsf{kind}(t) = \mathsf{in}$ if and only if there is a sub-state s_i in S such that $\mathsf{src}(t) \in A$ and $\mathsf{trg}(t) \in \mathsf{entries}(s_i)$.
4. $\mathsf{kind}(t) = \mathsf{out}$ if and only if there is a sub-state s_i in S such that $\mathsf{src}(t) \in \mathsf{exits}(s_i)$ and $\mathsf{trg}(t) \in B$.

Notation 2. In the remainder we will omit the entry and exit actions when $en = \perp$ and $ex = \perp$.

Example 2. Consider the State Machine depicted in Figure 2.[5] In our syntax this State Machine is described by the term s_1 where:

$$\begin{array}{rcl}
s_1 & \overset{def}{=} & [n_1, \{\mathsf{den}_{n_1}, p_1\}, \{\mathsf{dex}_{n_1}, q_1\}, \langle s_2, s_3\rangle, 1, \{t_1, t_2, t_3\}] \\
s_2 & \overset{def}{=} & [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}, q_2\}] \\
s_3 & \overset{def}{=} & [n_3, \{\mathsf{den}_{n_3}, p_2, p_3\}, \{\mathsf{dex}_{n_3}, q_3\}]
\end{array}$$

with transitions

$$\begin{array}{rcl}
t_1 & \overset{def}{=} & (\mathsf{sib}, \mathsf{true}, q_2, p_2, y, a_1) \\
t_2 & \overset{def}{=} & (\mathsf{in}, \mathsf{false}, p_1, p_3, \perp, \perp) \\
t_3 & \overset{def}{=} & (\mathsf{out}, \mathsf{true}, q_3, q_1, x, \perp)
\end{array}$$

Example 3. The State Machine in Figure 3 shows another example.

[5]In State Machine diagrams we label transitions $t_i : x_i/a_i$ where t_i is the name of the transition, x_i is the trigger event and a_i is the action. Each of these items can be omitted from the transition.

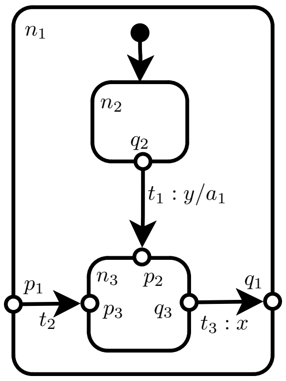

Figure 2: A simple State Machine.

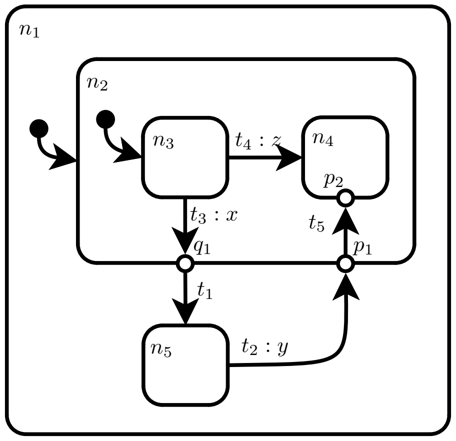

Figure 3: A State Machine with composite states.

This is encoded in our syntax as follows:

$$
\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_5\rangle, 1, \{t_1, t_2\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}, p_1\}, \{\mathsf{dex}_{n_2}, q_1\}, \langle s_3, s_4\rangle, 1, \{t_3, t_4, t_5\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}, p_2\}, \{\mathsf{dex}_{n_4}\}] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}\}]
\end{aligned}
$$

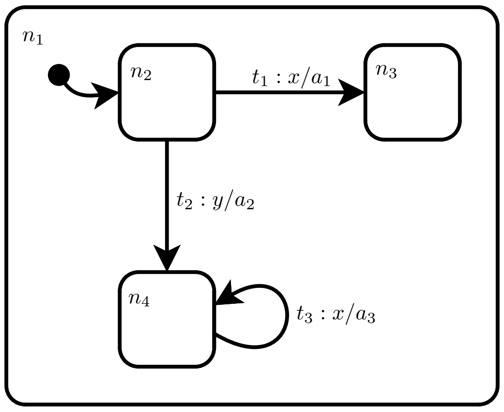

Figure 4: A simple State Machine.

with transitions

$$\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{false}, q_1, \mathsf{den}_{n_5}, \bot, \bot)\\
t_2 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_5}, p_1, y, \bot)\\
t_3 &\overset{def}{=} (\mathsf{out}, \mathsf{true}, \mathsf{dex}_{n_3}, q_1, x, \bot)\\
t_4 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_3}, \mathsf{den}_{n_4}, z, \bot)\\
t_5 &\overset{def}{=} (\mathsf{in}, \mathsf{false}, p_1, p_2, \bot, \bot)
\end{aligned}$$

4 The map

We begin assuming that states and transitions have no actions. We will incorporate actions later in Subsection 4.8.

In the remaining we will write $\mathsf{SM}_\top$ for $\mathsf{SM} \uplus \{\top\}$ where $\top$ denotes the "root", this is, the parent of the topmost state.

4.1 Representing hierarchical state structure

We begin by describing how the nesting order of a State Machine is represented in π_{klt}. The essence of the idea is to encode states as process definitions and we obtain the hierarchical structure via nested process definitions. Each state n_i will be encoded as a process definition named Sn_i.

For now we will ignore transitions in a State Machine, and basic states will have no behaviour and therefore will be mapped to the null process $\surd$. The main body of a process representing a composite state will simply invoke the process which corresponds to its default state.

We first show some examples and then the formalization of the map.

Example 4. Consider the State Machine from Figure 4. In our State Machine syntax this would be written as:

$$\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_3, s_4\rangle, 1, \{t_1, t_2, t_3\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}\}, \{\mathsf{dex}_{n_4}\}]
\end{aligned}$$

with transitions

$$\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_2}, \mathsf{den}_{n_3}, x, a_1) \\
t_2 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_2}, \mathsf{den}_{n_4}, y, a_2) \\
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_4}, x, a_3)
\end{aligned}$$

This would be represented as the following π_{klt} definition:

```
proc $Sn_1()$ = def {
   proc $Sn_2() = \surd$;
   proc $Sn_3() = \surd$;
   proc $Sn_4() = \surd$
} in $Sn_2()$
```

Here we have a process definition for the State Machine with a nested definition for each sub-state. The body of the main state invokes the process which corresponds to the default sub-state Sn_2.

Example 5. Now consider the State Machine from Figure 5. In our State Machine syntax this would be written as:

$$\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_3\rangle, 1, \{t_1\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}, \langle s_4, s_5\rangle, 1, \{t_2\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}\}, \{\mathsf{dex}_{n_4}\}] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}\}]
\end{aligned}$$

with transitions

$$\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_2}, \mathsf{den}_{n_3}, x, a_1) \\
t_2 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_5}, y, a_2)
\end{aligned}$$

This would be represented as the following π_{klt} definition:

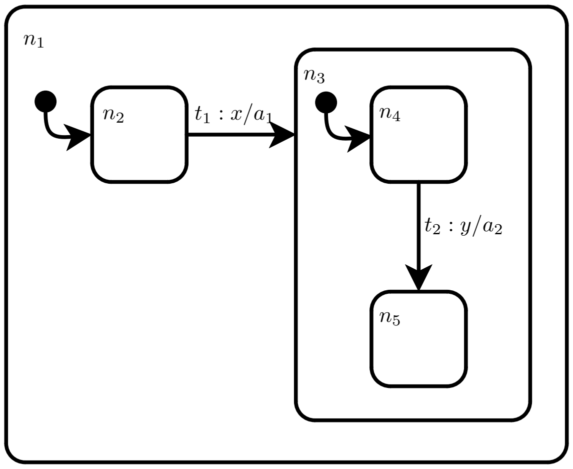

Figure 5: A simple State Machine.

```
proc $Sn_1() =$ def {
    proc $Sn_2() = \surd$;
    proc $Sn_3() =$ def {
        proc $Sn_4() = \surd$;
        proc $Sn_5() = \surd$
    } in $Sn_4()$;
} in $Sn_2()$
```

We now define a mapping $\mathcal{T}_0[\![\cdot]\!] : \mathsf{SM} \to \mathsf{KLT}$ for this encoding.

Definition 4. (Encoding nesting order)

$\mathcal{T}_0[\![[n, A, B, en, ex]]\!] \overset{def}{=} \mathsf{proc}\ Sn() = \surd$

$\mathcal{T}_0[\![[n, A, B, S, d, T, en, ex]]\!] \overset{def}{=} \mathsf{proc}\ Sn() = \mathsf{def}\ \{D_1; ...; D_k\}\ \mathsf{in}\ Sn_d()$

where each D_i is $\mathcal{T}_0[\![s_i]\!]$ for each s_i in $S = s_{1..k}$, and $n_d = \mathsf{name}(s_d)$ is the name of the default sub-state.

4.2 Representing basic (sibling) transitions

The concept of a transition in a State Machine represents the notion of state change when an event occurs. Since we represent states as processes, a state with transitions coming out of it is naturally represented by a listener process which waits for the appropriate events, and then becomes the process corresponding to the target of the transition.

In order to do this, the process definition of a state needs to know to which events it can respond. Such events would be encoded in the process interface (its ports), but because of lexical scoping in π_{klt} we only need to specify such events at the top level of the State Machine. In fact, we will later define a "wrapper" for the whole State Machine, which specifies its events. This allows us to simplify the encoding as all process definitions will have access to those events from the enclosing scope. For the time being we will not show the wrapper process, to simplify the presentation.

At this point we will consider only non-group transitions, this is, transitions whose source state is a basic state. Furthermore, we also ignore actions for now.

Example 6. Consider again the State Machine from Figure 4. This would be represented as the following π_{klt} definition:

$$\begin{array}{l}
\mathsf{proc}\ Sn_1() = \mathsf{def}\ \{ \\
\quad \mathsf{proc}\ Sn_2() = x?.Sn_3() + y?.Sn_4(); \\
\quad \mathsf{proc}\ Sn_3() = \surd; \\
\quad \mathsf{proc}\ Sn_4() = x?.Sn_4() \\
\}\ \mathsf{in}\ Sn_2()
\end{array}$$

Here the machine can go from state n_2 to n_3 when event x is triggered, and to n_4 when event y is triggered. This is naturally represented by a listener process $x?.Sn_3() + y?.Sn_4()$ which provides those choices. In state n_4 it can go back to itself when x is triggered, hence the definition Sn_4 is recursive.

Example 7. Now consider the State Machine from Figure 5. This would be represented as the following π_{klt} definition:

$$\begin{array}{l}
\mathsf{proc}\ Sn_1() = \mathsf{def}\ \{ \\
\quad \mathsf{proc}\ Sn_2() = x?.Sn_3(); \\
\quad \mathsf{proc}\ Sn_3() = \mathsf{def}\ \{ \\
\quad\quad \mathsf{proc}\ Sn_4() = y?.Sn_5(); \\
\quad\quad \mathsf{proc}\ Sn_5() = \surd \\
\quad \}\ \mathsf{in}\ Sn_4(); \\
\}\ \mathsf{in}\ Sn_2()
\end{array}$$

We now define a map $\mathcal{T}_1[\![\cdot]\!] : \mathsf{SM} \to \mathsf{SM}_\top \to \mathsf{KLT}$, which takes as input the State Machine term, and its enclosing state and returns the corresponding π_{klt} term.

Definition 5. (Encoding basic transitions)

- For a basic state $s \overset{def}{=} [n, A, B, en, ex]$ its translation is given by:

  $$\mathcal{T}_1[\![s]\!]_{s'} \overset{def}{=} \mathsf{proc}\ Sn() = \sum_{t_i \in T''} x_i?.Sn_i()$$

  where s's enclosing state is

  $$s' = [n', A', B', S', d', T', en', ex']$$

  and

  $$T'' \overset{def}{=} \{t \in T' \mid \exists q \in B.\, q = \mathsf{src}(t)\}$$

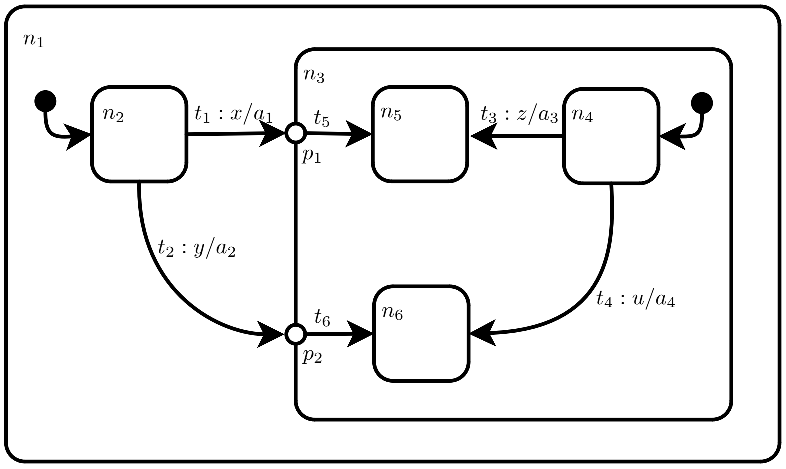

Figure 6: A State Machine with entry points.

is the set of transitions from T' whose source (q) is an exit point of state n;

$$x_i \overset{def}{=} \mathsf{evt}(t_i)$$

is the trigger event of transition t_i in the set T''; and

$$n_i = \mathsf{name}(s_i)$$

is the name of the target state $s_i \in S'$ with $\mathsf{trg}(t_i) \in \mathsf{entries}(s_i)$ (the target of the transition must be an entry point of the target state s_i).

- For a composite state $s = [n, A, B, S, d, T, en, ex]$ (with enclosing state s'):

$$\mathcal{T}_1[\![s]\!]_{s'} \overset{def}{=} \mathsf{proc}\ Sn() = \mathsf{def}\ \{D_1; ...; D_k\}\ \mathsf{in}\ Sn_d()$$

  where each D_i is $\mathcal{T}_1[\![s_i]\!]_s$ for each s_i in $S = s_{1..k}$ and $n_d = \mathsf{name}(s_d)$. Note that in this case the parameter passed to the translation of the sub-states is s, the composite state being translated. It is not necessary to pass the enclosing state s', since in UML-RT, transitions cannot cross boundaries. The state s' will be used later when we deal with group transitions.

4.3 Representing entry points and incoming transitions

We represent entry points of a state via a parameter of the process definition for that state. When the state is entered, this parameter is used to invoke the sub-state connected to the corresponding entry point.

Example 8. Consider the State Machine from Figure 6. In our syntax, this is written as:

$$
\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_3\rangle, 1, \{t_1, t_2\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}, p_1, p_2\}, \{\mathsf{dex}_{n_3}\}, \langle s_4, s_5, s_6\rangle, 1, \{t_3, t_4, t_5, t_6\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}\}, \{\mathsf{dex}_{n_4}\}] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}\}] \\
s_6 &\overset{def}{=} [n_6, \{\mathsf{den}_{n_6}\}, \{\mathsf{dex}_{n_6}\}]
\end{aligned}
$$

with transitions

$$
\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_2}, p_1, x, a_1) \\
t_2 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_2}, p_2, y, a_2) \\
t_3 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_5}, z, a_3) \\
t_4 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_6}, u, a_4) \\
t_5 &\overset{def}{=} (\mathsf{in}, \mathsf{false}, p_1, \mathsf{den}_{n_5}, \perp, \perp) \\
t_6 &\overset{def}{=} (\mathsf{in}, \mathsf{false}, p_2, \mathsf{den}_{n_6}, \perp, \perp)
\end{aligned}
$$

We encode this State Machine as follows:

```
proc $Sn_1(enp)$ = def {
    proc $Sn_2(enp) = x?.Sn_3(p_1) + y?.Sn_3(p_2)$;
    proc $Sn_3(enp)$ = def {
        proc $Sn_4(enp) = z?.Sn_5(\mathsf{den}_{n_5}) + u?.Sn_6(\mathsf{den}_{n_6})$;
        proc $Sn_5(enp) = \surd$;
        proc $Sn_6(enp) = \surd$
        proc $C(enp)$ =
            if $enp = p_1$ then $Sn_5(\mathsf{den}_{n_5})$
            else if $enp = p_2$ then $Sn_6(\mathsf{den}_{n_6})$
            else $Sn_4(\mathsf{den}_{n_4})$
    } in
        $C(enp)$
} in $Sn_2(\mathsf{den}_{n_2})$
```

We see that all definitions have a parameter *enp* which represents the entry point. The body of the definition Sn_3 contains a "dispatcher" C which makes the decision of which sub-state must be activated depending on the value of this parameter. If the parameter is none of the named entry points, it executed the process of the default sub-state. Note that transitions whose target is not a named entry point simply pass as parameter the default entry point den_{n_i}.

Now define a map $\mathcal{T}_2[\![\cdot]\!] : \mathsf{SM} \to \mathsf{SM}_\top \to \mathsf{KLT}$, which takes as input the State Machine term, and its enclosing state, and returns the corresponding π_{klt} term.

For simplicity and uniformity we give all states a parameter *enp*. In he case of basic states we simply ignore it.

Definition 6. (Encoding entry points)

- For a basic state $s \overset{def}{=} [n, A, B, en, ex]$ its translation is given by:

  $$\mathcal{T}_2[\![s]\!]_{s'} \overset{def}{=} \mathsf{proc}\ Sn(enp) = \sum_{t_i \in T''} x_i?.Sn_i(p_i)$$

  where s's enclosing state is

  $$s' = [n', A', B', S', d', T', en', ex']$$

  and

  $$T'' \overset{def}{=} \{t \in T' \mid \exists q \in B.\, q = \mathsf{src}(t)\}$$

  is the set of transitions from T' whose source (q) is an exit point of state n;

  $$x_i \overset{def}{=} \mathsf{evt}(t_i)$$

  is the trigger event of transition t_i in the set T''; and

  $$n_i = \mathsf{name}(s_i)$$

  is the name of the target state $s_i \in S'$ with $\mathsf{trg}(t_i) \in \mathsf{entries}(s_i)$ (the target of the transition must be an entry point of the target state s_i); and

  $$p_i \overset{def}{=} \mathsf{trg}(t_i)$$

  is the (name of the) entry point of the transition's target.

- For a composite state $s = [n, A, B, S, d, T, en, ex]$ (with enclosing state s'):

  $$\mathcal{T}_2[\![s]\!]_{s'} \overset{def}{=} \mathsf{proc}\ Sn(enp) = \mathsf{def}\ \{D_1; ...; D_k; C_{def}\}\ \mathsf{in}\ \ C(enp)$$

  where each D_i is $\mathcal{T}_2[\![s_i]\!]_s$ for each s_i in $S = s_{1..k}$ and C_{def} is the dispatcher defined as follows:

```
proc $C(enp) =$
    if        $enp = p_1$    then   $Sn_1(p'_1)$
    else if   $enp = p_2$    then   $Sn_2(p'_2)$
    ...
    else if   $enp = p_m$    then   $Sn_m(p'_m)$
    else      $Sn_d(p'_d)$.
```

where each $p_i \in A$ is a named entry point of s connected to the entry point p'_i of a sub-state n_i via an incoming transition $t_i = (\mathsf{in}, \mathsf{false}, p_i, p'_i, \bot, \bot) \in T$. Here we assume that the default state is n_d, with the initial transition connected to the entry point p'_d.

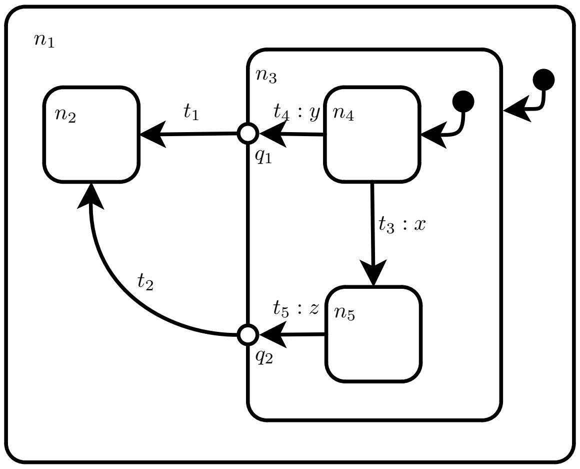

Figure 7: A State Machine with exit points.

4.4 Representing exit points

Exit points are simpler to represent than entry points. For basic states we don't need to represent them at all. For composite states, we represent them if there is an outgoing transition from some sub-state to the exit point. The exit point acts as a pseudo-state, an intermediate point between the actual source of the transition chain and its actual target. Hence, in our representation we create a simple process that immediately jumps to the destination.We use the convention of naming the definition for an exit point q as Bq. Strictly speaking we could avoid this definition, but having them explicitly makes the generated π_{klt} term easier to read.

Example 9. Consider the State Machine in Figure 7. Its representation is:

$$
\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_3\rangle, 2, \{t_1, t_2\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}, \langle s_4, s_5\rangle, 1, \{t_3, t_4, t_5\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}\}, \{\mathsf{dex}_{n_4}\}] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}\}]
\end{aligned}
$$

with transitions

$$\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{false}, q_1, \mathsf{den}_{n_2}, \bot, \bot)\\
t_2 &\overset{def}{=} (\mathsf{sib}, \mathsf{false}, q_2, \mathsf{den}_{n_2}, \bot, \bot)\\
t_3 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_5}, x, \bot)\\
t_4 &\overset{def}{=} (\mathsf{out}, \mathsf{true}, \mathsf{dex}_{n_4}, q_1, y, \bot)\\
t_5 &\overset{def}{=} (\mathsf{out}, \mathsf{true}, \mathsf{dex}_{n_5}, q_2, z, \bot)
\end{aligned}$$

In π_{klt} this is represented as:[6]

```
proc $Sn_1(enp)$ = def {
    proc $Sn_2(enp) = \surd$;
    proc $Sn_3(enp)$ = def {
        proc $Sn_4(enp) = x?.Sn_5(\mathsf{den}_{n_5}) + y?.Bq_1()$;
        proc $Sn_5(enp) = z?.Bq_2()$;
        proc $Bq_1() = Sn_2(\mathsf{den}_{n_2})$;
        proc $Bq_2() = Sn_2(\mathsf{den}_{n_2})$
    } in $Sn_4(\mathsf{den}_{n_4})$
} in $Sn_3(\mathsf{den}_{n_3})$
```

Now define a map $\mathcal{T}_3[\![\cdot]\!] : \mathsf{SM} \to \mathsf{SM}_\top \to \mathsf{KLT}$, which takes as input the State Machine term, and its enclosing state, and returns the corresponding π_{klt} term. For simplicity and uniformity we give all states a parameter *enp*. In he case of basic states we simply ignore it.

Definition 7. (Encoding exit points)

- For a basic state $s \overset{def}{=} [n, A, B, en, ex]$ its translation is given by:

  $$\mathcal{T}_3[\![s]\!]_{s'} \overset{def}{=} \mathsf{proc}\ Sn(enp) = \sum_{t_i \in T''} x_i?.Q_i$$

  where s's enclosing state is

  $$s' = [n', A', B', S', d', T', en', ex']$$

  and

  $$T'' \overset{def}{=} \{t \in T' \mid \exists q \in B.\, q = \mathsf{src}(t)\}$$

  is the set of transitions from T' whose source (q) is an exit point of state n;

  $$x_i \overset{def}{=} \mathsf{evt}(t_i)$$

[6]In the remainder we will obviate the dispatcher C of a composite state if it only has the default entry point and no named entry points.

is the trigger event of transition t_i in the set T''; and Q_i is the target of the transition, defined as

$$Q_i \overset{def}{=} \begin{cases} Sn_i(p_i) & \text{if } \mathsf{kind}(t_i) = \mathsf{sib},\ \mathsf{trg}(t_i) = p_i \\ & \text{and } \exists s_i \in S'.\, p_i \in \mathsf{entries}(s_i) \text{ and } n_i = \mathsf{name}(s_i) \\ Bq_i() & \text{if } \mathsf{kind}(t_i) = \mathsf{out} \text{ and } \mathsf{trg}(t_i) = q_i \in B' \end{cases}$$

- For a composite state $s = [n, A, B, S, d, T, en, ex]$ (with enclosing state s'):

  $$\mathcal{T}_3[\![s]\!]_{s'} \overset{def}{=} \mathsf{proc}\ Sn(enp) = \mathsf{def}\ \{D_1; ...; D_k; B_1; ...; B_m; C_{def}\}\ \mathsf{in}\ \ C(enp)$$

  where each D_i is $\mathcal{T}_3[\![s_i]\!]_s$ for each s_i in $S = s_{1..k}$, each B_i is a process definition for exit point $q_i \in B$, given by

  $$B_i \overset{def}{=} \mathsf{proc}\ Bq_i() = Q_j$$

  where Q_j is the target of the exit point, and is defined as above, and C_{def} is the dispatcher defined as follows:

```
proc $C(enp)$ =
   if        $enp = p_1$   then   $Sn_1(p'_1)$
   else if   $enp = p_2$   then   $Sn_2(p'_2)$
   ...
   else if   $enp = p_m$   then   $Sn_m(p'_m)$
   else      $Sn_d(p'_d)$.
```

where each $p_i \in A$ is a named entry point of s connected to the entry point p'_i of a sub-state n_i via an incoming transition $t_i = (\mathsf{in}, \mathsf{false}, p_i, p'_i, \bot, \bot) \in T$. Here we assume that the default state is n_d, with the initial transition connected to the entry point p'_d.

4.5 Representing group transitions

Group transitions are transitions whose source state is a composite state. When a group transition is triggered, the source state and all its sub-states are exited. Hence, in a sense, a group transition acts as an interrupt on the current state.

Traditionally, group transitions are interpreted by flattening the State Machine and adding a corresponding transition to every sub-state of the group transition's source. We take a different approach to preserve modularity. First, we add in every sub-state a pair of events *exit* and *exack*, which are used, respectively, to tell the state to exit, and to acknowledge the exit from that state. Second, in the composite state that is the source of the group transition we add an *event handler*, whose job is to listen for events (the triggers of the group transitions) and whenever one such event occurs, tells its currently active sub-state to exit and then waits for the sub-state to acknowledge the exit before jumping to the actual destination. Waiting for the sub-state to exit ensures that the sequence of exit actions will be executed in the correct order.

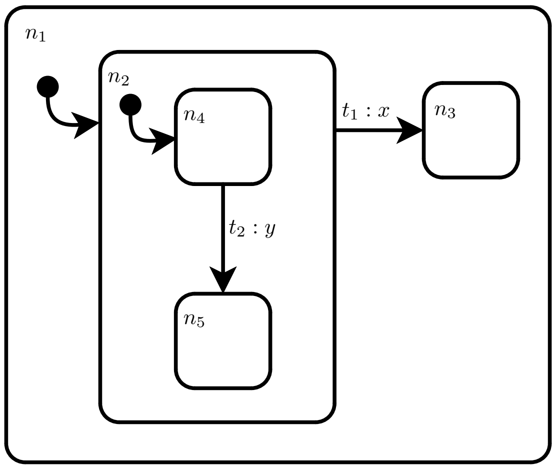

Figure 8: A State Machine with a group transition.

Example 10. Consider the State Machine from Figure 8. This is represented as:

$$
\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_3\rangle, 1, \{t_1\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}\}, \langle s_4, s_5\rangle, 1, \{t_2\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}\}, \{\mathsf{dex}_{n_4}\}] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}\}]
\end{aligned}
$$

with transitions

$$
\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_2}, \mathsf{den}_{n_3}, x, \perp) \\
t_2 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_5}, y, \perp)
\end{aligned}
$$

In π_{klt} this would be represented as:

```
proc $Sn_1(enp)$ = def {
    proc $Sn_2(enp)$ = def {
        proc $Sn_4(enp, exit, exack) = y?.Sn_5(\mathsf{den}_{n_5}, exit, exack) + exit?.exack!;$
        proc $Sn_5(enp, exit, exack) = exit?.exack!;$
        proc $H(exit', exack') = x?.exit'!.exack'?.Sn_3(\mathsf{den}_{n_3})$
    } in new $exit', exack'$ in $(Sn_4(\mathsf{den}_{n_4}, exit', exack') \parallel H(exit', exack'));$
    proc $Sn_3(enp) = \surd;$
} in $Sn_2(\mathsf{den}_{n_2})$
```

Note that we have added ports *exit* and *exack* to the process definitions for sub-states n_4 and n_5, and in their bodies, the event listener has a branch

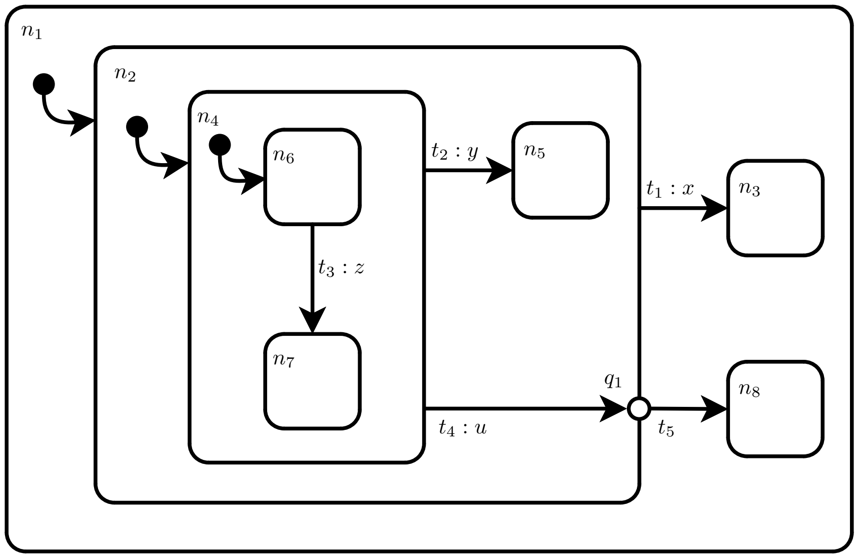

Figure 9: A State Machine with a group transition.

$exit?.exack!$, which when it receives an $exit$ event from the containing state, answers with $exack$ (exit acknowledgment) and stops. In addition to these sub-states, Sn_2 contains a definition for the event handler H with ports $exit'$ and $exack'$ which are the events linking it to the current sub-state: when the event x of the group transition t_1 is triggered, H tells its current sub-state to exit by triggering $exit'$ and waits for the acknowledgment $exack'$. Once it gets the acknowledgment, it can jump to the destination Sn_3. The main body of Sn_2 is $\mathsf{new}\ exit', exack'\ \mathsf{in}\ (Sn_4(\mathsf{den}_{n_4}, exit', exack') \parallel H(exit', exack'))$. Here Sn_2 creates two local events $exit'$ and $exack'$ to communicate with its currently active sub-state. Then it launches its default sub-state and the event handler. Note that the sub-states Sn_4 and Sn_5 are both passed the events $exit'$ and $exack'$ when invoked. This way, the sub-states are connected to their container's event handler H.

Now we extend this example further to show how composite states themselves should handle exit messages.

Example 11. Consider the State Machine from Figure 9. This example is as the previous one, with state n_4 changed from a basic state to a composite state containing sub-states n_6 and n_7, and adding an additional transition chain t_4, t_5

to state n_8. This is represented as:

$$
\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_3, s_8\rangle, 1, \{t_1, t_5\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}, q_1\}, \langle s_4, s_5\rangle, 1, \{t_2, t_4\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}\}, \{\mathsf{dex}_{n_4}\}, \langle s_6, s_7\rangle, 1, \{t_3\}] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}\}] \\
s_6 &\overset{def}{=} [n_6, \{\mathsf{den}_{n_6}\}, \{\mathsf{dex}_{n_6}\}] \\
s_7 &\overset{def}{=} [n_7, \{\mathsf{den}_{n_7}\}, \{\mathsf{dex}_{n_7}\}] \\
s_8 &\overset{def}{=} [n_8, \{\mathsf{den}_{n_8}\}, \{\mathsf{dex}_{n_8}\}]
\end{aligned}
$$

with transitions

$$
\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_2}, \mathsf{den}_{n_3}, x, \bot) \\
t_2 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_5}, y, \bot) \\
t_3 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_6}, \mathsf{den}_{n_7}, z, \bot) \\
t_4 &\overset{def}{=} (\mathsf{out}, \mathsf{true}, \mathsf{dex}_{n_4}, q_1, u, \bot) \\
t_5 &\overset{def}{=} (\mathsf{sib}, \mathsf{false}, q_1, \mathsf{den}_{n_8}, \bot, \bot)
\end{aligned}
$$

In π_{klt} this would be represented as:

```
proc $Sn_1(enp)$ = def {
    proc $Sn_2(enp)$ = def {
        proc $Sn_4(enp, exit, exack, sh)$ = def {
            proc $Sn_6(enp, exit, exack) =$
                $z?.Sn_7(\mathsf{den}_{n_7}, exit, exack) + exit?.exack!;$
            proc $Sn_7(enp, exit, exack) = exit?.exack!;$
            proc $H(exit', exack') =$
                $y?.exit'!.exack'?.Sn_5(\mathsf{den}_{n_5}, exit, exack, sh)$
                $+u?.exit'.exack'?.Bq_1(sh)$
                $+exit?.exit'!.exack'?.exack!$
        } in new $exit', exack'$ in $(Sn_6(\mathsf{den}_{n_6}, exit', exack') \parallel H(exit', exack'));$
        proc $Sn_5(enp, exit, exack, sh) = exit?.exack!;$
        proc $Bq_1(sh') = sh'! \parallel Sn_8(\mathsf{den}_{n_8});$
        proc $H(exit', exack', sh') = x?.exit'!.exack'?.Sn_3(\mathsf{den}_{n_3}) + sh'?.\surd$
    } in
        new $exit', exack', sh'$ in
            $(Sn_4(\mathsf{den}_{n_4}, exit', exack', sh') \parallel H(exit', exack', sh'));$
    proc $Sn_3(enp) = \surd;$
    proc $Sn_8(enp) = \surd;$
} in $Sn_3(\mathsf{den}_{n_3})$
```

Here we have the same concept, but the main difference is in the event handler H of Sn_4. This event handler can receive either a y event, a u event or an *exit* request from the containing state's handler (Sn_2's handler). Hence, if the machine is in state Sn_4 (and therefore in state Sn_2 as well), and an event x arrives, Sn_2's handler sends an *exit* request to Sn_4, which is received by its own handler, which in turn sends an exit request to its currently active sub-state (Sn_6 or Sn_7). When the sub-state acknowledges, Sn_4's handler itself sends an exit acknowledgment to Sn_2's handler, which then jumps to Sn_3.

Note how in this example it is possible to exit from state n_2 via either the group transition t_1, or via the transition chain t_4, t_5 through exit point q_1. If the latter route is taken, then the handler H for Sn_2 must be terminated before exiting. In order to do this, the handler H has a port sh (stop handler) to stop it. The channel to connect this port to the sub-states is called sh' in Sn_2, and it is passed between sub-states, so that when exit point Bq_1 is executed, the event sh' is triggered, stopping the handler.

Now define a map $\mathcal{T}_4[\![\cdot]\!] : \mathsf{SM} \to \mathsf{SM}_\top \to \mathsf{KLT}$, which takes as input the State Machine term, and its enclosing state, and returns the corresponding π_{klt} term. For simplicity and uniformity we give all states parameters $enp, exit, exack$.

Definition 8. (Encoding group transitions)

- For a basic state $s \overset{def}{=} [n, A, B, en, ex]$ its translation is given by:

$$\mathcal{T}_4[\![s]\!]_{s'} \overset{def}{=} \mathsf{proc}\ Sn(enp, exit, exack) = \sum_{t_i \in T''} x_i?.Q_i + exit?.exack!$$

  where s's enclosing state is

$$s' = [n', A', B', S', d', T', en', ex']$$

  and

$$T'' \overset{def}{=} \{t \in T' \mid \exists q \in B.\, q = \mathsf{src}(t)\}$$

  is the set of transitions from T' whose source (q) is an exit point of state n;

$$x_i \overset{def}{=} \mathsf{evt}(t_i)$$

  is the trigger event of transition t_i in the set T''; and Q_i is the target of the transition, defined as

$$Q_i \overset{def}{=} \begin{cases} Sn_i(p_i, exit, exack, sh) & \text{if } \mathsf{kind}(t_i) = \mathsf{sib},\ \mathsf{trg}(t_i) = p_i \\ & \text{and } \exists s_i \in S'.\, p_i \in \mathsf{entries}(s_i) \text{ and } n_i = \mathsf{name}(s_i) \\ Bq_i(sh) & \text{if } \mathsf{kind}(t_i) = \mathsf{out} \text{ and } \mathsf{trg}(t_i) = q_i \in B' \end{cases}$$

- For a composite state $s = [n, A, B, S, d, T, en, ex]$ (with enclosing state s'):

$$
\begin{array}{l}
\mathcal{T}[\![s]\!]_{s'} \overset{def}{=} \\
\quad \textsf{proc } Sn(enp, exit, exack, sh) = \\
\quad\quad \textsf{def } \{D_1; ...; D_k; B_1; ...; B_l; C_{def}; H_{def}\} \textsf{ in} \\
\quad\quad\quad \textsf{new } \ exit', exack', sh' \textsf{ in} \\
\quad\quad\quad\quad (C(enp, exit', exack', sh') \\
\quad\quad\quad\quad\ \ \| \ H(exit', exack', sh')).
\end{array}
$$

where each D_i is $\mathcal{T}[\![s_i]\!]_s$ for each s_i in S ; each B_i is a process definition for exit point $q_i \in B$, given by

$$B_i \overset{def}{=} \textsf{proc } Bq_i(sh') = (sh'! \ \| \ Q_j).$$

where Q_j is the target of the exit point, and is defined as above; C_{def} is the dispatcher defined as follows:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textsf{proc } C(enp, exit, exack, sh) =} \\
\quad \textsf{if} & enp = p_1 & \textsf{then} & Sn_1(p_1', exit', exack', sh') \\
\quad \textsf{else if} & enp = p_2 & \textsf{then} & Sn_2(p_2', exit', exack', sh') \\
\quad \ldots & & & \\
\quad \textsf{else if} & enp = p_m & \textsf{then} & Sn_m(p_m', exit', exack', sh') \\
\quad \textsf{else} & \multicolumn{3}{l}{Sn_d(p_d', exit', exack', sh').}
\end{array}
$$

where each $p_i \in A$ is a named entry point of s connected to the entry point p_i' of a sub-state n_i via an incoming transition $t_i = (\textsf{in}, \textsf{false}, p_i, p_i', \bot, \bot) \in T$. Here we assume that the default state is n_d, with the initial transition connected to the entry point p_d'. Finally, H_{def} is the definition of the event handler H, as follows:

$$
\begin{array}{l}
\textsf{proc } H(exit', exack', sh') = \\
\quad \sum_{t_i \in T''} x_i?.exit'!.exack'?.Q_i \\
\quad + exit?.exit'!.exack'?.exack! \ + \ sh'?.\surd.
\end{array}
$$

where T'' and Q_i are defined as for basic states. [7]

4.6 Enabled-transition selection policy

It is possible that two transitions are simultaneously enabled if their source is the currently active state and they share the same trigger event. In this case the transitions are said to be in conflict. If the source of one such transition

[7]Note that in H_{def} and C_{def} the (non-primed) *exit* and *exack* events are those which are used to interact with the containing state, while the (primed) $exit'$ and $exack'$ events are used to interact with the currently active sub-state. Also note that it is not necessary for H to have explicit parameters for *exit* and *exack* due to the lexical scoping rules of π_{klt}, as H_{def} is inside the definition of its containing state. This allows the handler itself to receive exit requests from its own containing state.

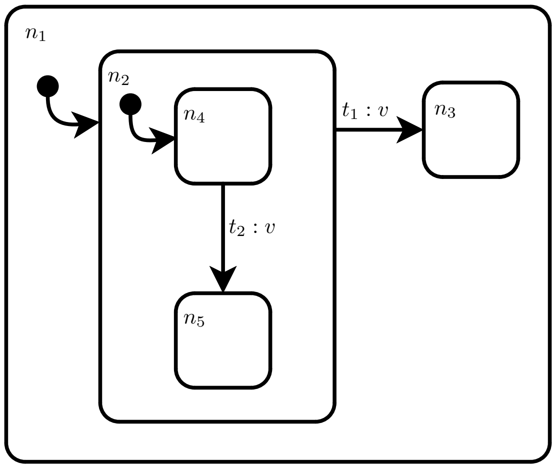

Figure 10: A State Machine with a group transition.

is a sub-state of the source of the other transition, then the conflict is resolved by giving priority to the former, inner transition. In this section we implement such priority scheme.[8]

The main idea is as follows. For each composite state n, the handler receives the incoming event and before it compares it with the triggers of the transitions from n, it forwards the event "down" to its currently active sub-state n'. If n' (or a sub-state) has a transition with this event then it handles the event and sends an "accepted" message back to n's handler. On the other hand, if n' (or a sub-state) didn't have such a transition, then it sends a "rejected" message back to n's handler. If n's handler receives from n' an "accepted" message, it in turn sends an "accepted" message to its containing state. If it receives a "rejected" message, it compares the event with the triggers of n's transitions. If one trigger matches, an "accepted" message is sent to the containing state of n and the transition is taken. Otherwise, a "rejected" message is sent.

In order to implement this, we modify our current translation so that instead of treating State Machine events as π_{klt} events, we give each process definition a port *inp* where the input event will arrive. Hence, rather than representing a basic transition with trigger x and target n_i as $x?.Sn_i(...)$, we will represent it as $inp?\text{“}x\text{”}.Sn_i(...)$. We also add an *acc* and a *rej* port to inform the containing state of acceptance or rejection of events.

Example 12. Consider the State Machine from Figure 10. where if the current state is n_4 we have that on event v, both transitions t_1 and t_2 are enabled and therefore in conflict.

In π_{klt} this example would be represented as follows[9]:

[8]Note that this "priority" is different from the priority of events in the event queue. Such event priorities will be addressed later.

[9]In this example we are abstracting the handler for n_1 for the sake of simplicity, but it

```
proc $Sn_1(inp, acc, rej, enp)$ = def {
  proc $Sn_2(inp, acc, rej, enp)$ = def {
    proc $Sn_4(inp, acc, rej, enp, exit, exack)$ =
      $inp?x.$if $x = $ “$v$” then $acc!.Sn_5(inp, acc, rej, \mathsf{den}_{n_5}, exit, exack)$
                              else $rej!.Sn_4(inp, acc, rej, enp, exit, exack)$
      $+exit?.exack!$;
    proc $Sn_5(inp, acc, rej, enp, exit, exack)$ =
      $inp?x.rej!.Sn_5(inp, acc, rej, enp, exit, exack) + exit?.exack!$;
    proc $H(inp', acc', rej', exit', exack')$ =
      $inp?x.inp'!x.(acc'?.acc!.H(inp', acc', rej', exit', exack')$
                   $+ rej'?.$
                        if $x = $ “$v$”
                          then $exit'!.exack'?.acc!.Sn_3(inp, acc, rej, enp)$
                          else $rej!.H(inp', acc', rej', exit', exack'))$
      $+exit?.exit!.exack'?.exack!$;
  }
    in
      new $inp', acc', rej', exit', exack'$ in
        $(Sn_4(inp', acc', rej', \mathsf{den}_{n_4}, exit', exack')$
         $\| H(inp' acc', rej', exit', exack'))$;
  proc $Sn_3(inp, acc, rej, enp) = \surd$;
} in $Sn_2(inp, acc, rej, \mathsf{den}_{n_2})$
```

Here, each process definition is extended with three additional ports, *inp*, *acc* and *rej*, as explained above. Each composite process (such as Sn_2) creates channels inp', acc' and rej' to interact with its sub-states (as well as $exit'$ and $exack'$ as explained in Subsection 4.5). These are then passed to both the current sub-state and the handler H.

State Sn_4 waits for an event x ($inp?x$) and when it arrives, it compares it with its outgoing transitions (only one in this case). For each possible matching trigger, it triggers the accept event ($acc!$) and then performs the transition to its target. The last case of the conditional is when there is no matching trigger, in which case it triggers the reject event ($rej!$) and remains in the same state. Additionally if an *exit* event arrives, it acknowledges it and stops ($exit?.exack!$).

The behaviour of state Sn_5 is similar to Sn_4, except that all events arriving on the *inp* port are rejected as this state has no outgoing transitions.

The handler for Sn_2, like any other state, either accepts an input event ($inp?x$) or an exit request ($exit?$). If it is an input request, it forwards it down to its currently active sub-state ($inp'!$) and waits for the sub-state to accept it ($acc'?$) or reject it ($rej'?$). If it was accepted, it forwards an accept to the enclosing state ($acc!$). If it was rejected, then, as with Sn_4, the event is matched against the trigger of each transition whose source is n_2, which in this example is only one. For each matching transition, an exit event is sent to the currently active sub-state ($exit'!$) and an acknowledgment is expected ($exack'?$), then an accept signal is sent to the containing state ($acc!$) and the transition

would be analogous to that of n_2.

is performed. If no transition matched, a reject signal is sent to the containing state ($rej!$) and the handler goes back to waiting. Finally, if the handler received an exit request from its containing state, it sends an exit request to the currently active sub-state ($exit'!$) and an acknowledgment is expected ($exack'?$),ending with an acknowledgment to the containing state ($exack!$).

Now define a map $\mathcal{T}_5[\![\cdot]\!] : \mathsf{SM} \rightarrow \mathsf{SM}_\top \rightarrow \mathsf{KLT}$, which takes as input the State Machine term, and its enclosing state, and returns the corresponding π_{klt} term. For simplicity and uniformity we give all states parameters $inp, acc, rej, enp, exit, exack, sh$.

Definition 9. (Encoding group transitions with priorities)

- For a basic state $s \overset{def}{=} [n, A, B, en, ex]$ its translation is given by:

$$\begin{array}{l}
\mathcal{T}_5[\![s]\!]_{s'} \overset{def}{=} \\
\quad \mathsf{proc}\ Sn(inp, acc, rej, enp, exit, exack, sh) = \\
\quad\quad inp?x. \\
\quad\quad\quad \mathsf{if}\ x = \text{“}x_1\text{”}\ \mathsf{then}\ acc!.Q_1 \\
\quad\quad\quad \mathsf{else\ if}\ x = \text{“}x_2\text{”}\ \mathsf{then}\ acc!.Q_2 \\
\quad\quad\quad \ldots \\
\quad\quad\quad \mathsf{else\ if}\ x = \text{“}x_m\text{”}\ \mathsf{then}\ acc!.Q_m \\
\quad\quad\quad \mathsf{else}\ rej!.Sn(inp, acc, rej, enp, exit, exack, sh) \\
\quad\quad +exit?.exack!
\end{array}$$

  where s's enclosing state is

$$s' = [n', A', B', S', d', T', en', ex']$$

  and

$$T'' \overset{def}{=} \{t \in T' \mid \exists q \in B.\, q = \mathsf{src}(t)\}$$

  is the set of transitions from T' whose source (q) is an exit point of state n; each

$$x_i \overset{def}{=} \mathsf{evt}(t_i)$$

  is the trigger event of transition t_i in the set T''; and Q_i is the target of the transition, defined as

$$Q_i \overset{def}{=} \begin{cases} Sn_i(inp, acc, rej, p_i, exit, exack, sh) & \text{if } \mathsf{kind}(t_i) = \mathsf{sib},\ \mathsf{trg}(t_i) = p_i, \\ & \ \exists s_i \in S'.\, p_i \in \mathsf{entries}(s_i), \\ & \ \text{and } n_i = \mathsf{name}(s_i) \\ Bq_i(sh) & \text{if } \mathsf{kind}(t_i) = \mathsf{out} \\ & \ \text{and } \mathsf{trg}(t_i) = q_i \in B' \end{cases}$$

- For a composite state $s = [n, A, B, S, d, T, en, ex]$ (with enclosing state s'):

$$
\begin{array}{l}
\mathcal{T}_5[\![s]\!]_{s'} \overset{def}{=} \\
\quad \textsf{proc } Sn(inp, acc, rej, enp, exit, exack, sh) = \\
\quad\quad \textsf{def } \{D_1; ...; D_k; B_1; ...; B_l; C_{def}; H_{def}\} \textsf{ in} \\
\quad\quad\quad \textsf{new } inp', acc', rej', exit', exack', sh' \textsf{ in} \\
\quad\quad\quad\quad (C(inp', acc', rej', enp, exit', exack', sh') \\
\quad\quad\quad\quad\ \ \| \ H(inp', acc', rej', exit', exack', sh'))
\end{array}
$$

where each D_i is $\mathcal{T}_5[\![s_i]\!]_s$ for each s_i in $S = s_{1..k}$; each B_i is a process definition for exit point $q_i \in B$, given by[10]

$$B_i \overset{def}{=} \textsf{proc } Bq_i(sh') = sh'! \parallel Q_j$$

where Q_j is the target of the exit point, and is defined as above, and C_{def} is the dispatcher defined as follows:

```
proc $C(inp', acc', rej', enp, exit', exack', sh') =$
    if       $enp = p_1$    then    $Sn_1(inp', acc', rej', p'_1, exit', exack', sh')$
    else if  $enp = p_2$    then    $Sn_2(inp', acc', rej', p'_2, exit', exack', sh')$
    $\cdots$
    else if  $enp = p_m$    then    $Sn_m(inp', acc', rej', p'_m, exit', exack', sh')$
    else     $Sn_d(inp', acc', rej', p'_d, exit', exack', sh')$.
```

where each $p_i \in A$ is a named entry point of s connected to the entry point p'_i of a sub-state n_i via an incoming transition $t_i = (\textsf{in}, \textsf{false}, p_i, p'_i, \bot, \bot) \in T$. Here we assume that the default state is n_d, with the initial transition connected to the entry point p'_d. Finally, H_{def} is the definition of the event handler H, as follows:

```
proc $H(inp', acc', rej', exit', exack', sh') =$
    $inp?x.inp'!x.$
        $(acc'?.acc!.H(inp', acc', rej', exit', exack', sh')$
        $+rej'?.$
            if $x =$ “$x_1$” then $exit'!.exack'?.acc!.Q_1$
            else if $x =$ “$x_2$” then $exit'!.exack'?.acc!.Q_2$
            $\cdots$
            else if $x =$ “$x_m$” then $exit'!.exack'?.acc!.Q_m$
            else $rej!.H(inp', acc', rej', exit', exack', sh'))$
    $+exit?.exit'!.exack'?.exack!$
    $+sh'?.\surd$
```

where T'' and Q_i are defined as for basic states.

[10]Note that in the definition of Bq_i the parameter is sh' and not sh. This is because if q_i is connected through an out transition to some q_j in the enclosing state, then the process Q_j will be $Bq_j(sh)$ where sh is the signal to stop the parent's handler. Hence we must distinguish between the two.

4.7 History

Whenever a composite state is entered for the first time, its *initial* sub-state is entered. If, however, the composite state was previously visited, and the composite state is entered through an entry point not explicitly connected to any sub-state, it enters the last visited sub-state, i.e. the sub-state which was active when the composite state exited. This behaviour is called *history.* The policy applies recursively for the sub-state, resulting in what is known as *deep history.*

Our mapping so far ignores history so when a composite state is entered through an entry point not connected to a sub-state, the initial sub-state is entered, in other words, up until now, the default is the initial sub-state. In order to implement history we need to record the active sub-state when we exit the composite state so that we reactivate it the next time we enter. To implement this kind of memory we need operations to store data in some "memory cell" and to retrieve it later. We can model such operations in our calculus, using some syntactic sugar for readability:

$$\mathsf{set}\, x \;:=\; v \;\overset{def}{=}\; x?a.a!v$$

This stores the value v in a "memory cell" x. This cell expects a message on x which provides a channel a where the answer (the contents of the cell) is to be sent. Note that the cell is ephemeral: once it is read, its contents are lost.

$$\mathsf{let}\ v = \mathsf{get}\, x\ \mathsf{in}\ P \;\overset{def}{=}\; \nu a.x!a.a?v.P$$

This process retrieves the value v stored in cell x, by creating a private response channel a and sending it to the cell which provides the contents v which can be used in process P.

Now, to model history, every composite state n will have for each sub-state n_i, a pair of channels h_i (history) and r_i (reenter or reactivate). The channel h_i will be the memory cell which stores the last sub-state of n_i when n_i last exited.[11] The channel r_i represents the event of "reentering" state n_i. In fact, the value stored in h_i will be the r_k corresponding to to the sub-state n_k of n_i when n_i exited. Hence, when state n_i is reentered, it will retrieve from h_i the link r_k which will be used to reactivate the sub-state n_k. Since this policy is executed by n_k as well (if it is composite), the end result corresponds to the deep history policy of UML-RT.

More precisely, whenever we exit a sub-state n_k of a composite state n_i we do the following in the definition of process Sn_k:

$$\mathsf{set}\, h_i := r_k \parallel r_k?(enp, ...).Sn_k(enp, ...)$$

This is, first store the reactivation signal r_k in the parent's memory cell h_i. Then we set a "frozen" process which listens to the reactivation signal r_k, which expects the appropriate entry parameters.

[11]If n_i is a basic state, then h_i will be unused.

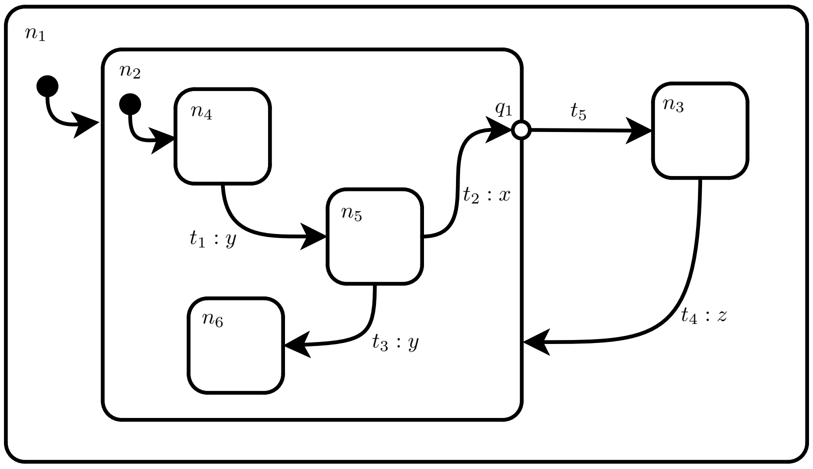

Figure 11: History

When we enter state n_i, if it is not a named entry point, or the initial "*init*" point, then we retrieve the signal stored in h_i and trigger it:

$$\mathsf{let}\ \ r = \mathsf{get}\, h_i\ \mathsf{in}\ r!(\text{"}hist\text{"}, ...)$$

The "*hist*" entry point passed tells the sub-state to recursively apply the same policy. Note that entering a state n_i, providing its default entry point den_{n_i} has the same effect as providing "*hist*" as entry point. Also note that, if we replace this by "*init*" we would obtain the so called *shallow-history* policy.

Example 13. Consider the State Machine in Figure 11. In this example, if the active sub-state is n_5 when an x event arrives, the transition chain t_2, t_5 is taken to n_3. If this is followed by a z event, transition t_4 is taken back to n_2 and into n_5, since it was the last active sub-state of n_2 when it was exited.

This is represented as:

$$\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_3\rangle, 1, \{t_4, t_5\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}, q_1\}, \langle s_4, s_5, s_6\rangle, 1, \{t_1, t_2, t_3\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}\}, \{\mathsf{dex}_{n_4}\}] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}\}] \\
s_6 &\overset{def}{=} [n_6, \{\mathsf{den}_{n_6}\}, \{\mathsf{dex}_{n_6}\}]
\end{aligned}$$

with transitions

$$\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_5}, y, \bot) \\
t_2 &\overset{def}{=} (\mathsf{out}, \mathsf{true}, \mathsf{dex}_{n_5}, q_1, x, \bot) \\
t_3 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_5}, \mathsf{den}_{n_6}, y, \bot) \\
t_4 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_3}, \mathsf{den}_{n_2}, z, \bot) \\
t_5 &\overset{def}{=} (\mathsf{sib}, \mathsf{false}, q_1, \mathsf{den}_{n_3}, \bot, \bot)
\end{aligned}$$

In this example, the State Machine first enters n_1, then n_2 and then n_4. On an input sequence y, x, z, y, it goes from n_4 to n_5, to n_3 then enters n_2 again and goes back to n_5, and finally to n_6.

We will first present the π_{klt} representation, for the sake of simplicity, without the mechanism for handling group transitions from Subsection 4.5 or conflicting transition resolution from Subsection 4.6:[12]

```
new $h_1, h_2, h_3, h_4, h_5, h_6, r_1, r_2, r_3, r_4, r_5, r_6$ in def {
  proc $Sn_1(enp)$ = def {
    proc $Sn_2(enp)$ = def {
      proc $Sn_4(enp) = y?.Sn_5(\mathsf{den}_{n_5})$;
      proc $Sn_5(enp)$ =
        $x?.(\mathsf{set}\, h_2 := r_5 \parallel r_5?enp.Sn_5(enp) \parallel Bq_1())$
        $+y?.Sn_6(\mathsf{den}_{n_6})$;
      proc $Bq_1() = Sn_3(\mathsf{den}_{n_3})$;
      proc $Sn_6(enp) = \surd$;
    } in
      if $enp$ = "$init$" then $Sn_4$("$init$")
      else let $r$ = get $h_2$ in $r!$("$hist$");
    proc $Sn_3(enp) = z?.Sn_2(\mathsf{den}_{n_2})$;
  } in $Sn_2$("$init$")
} in $Sn_1$("$init$")
```

Now we extend the example to see the effect of deep history by making state n_5 a composite state as shown in Figure 12. This is represented as:

[12]We also ignore history of the top-level state n_1 for the sake of simplicity.

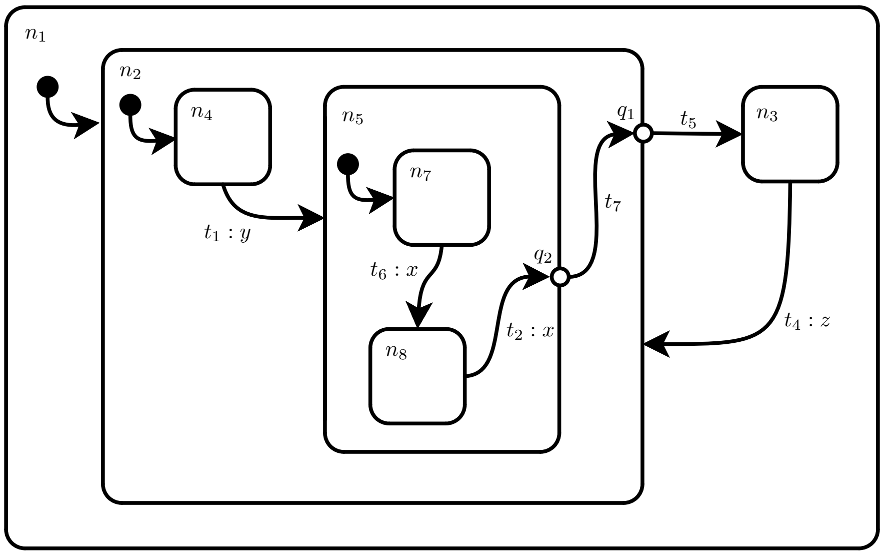

Figure 12: Deep history.

$$
\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_3 \rangle, 1, \{t_4, t_5\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}, q_1\}, \langle s_4, s_5, s_6 \rangle, 1, \{t_1, t_3, t_7\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}\}, \{\mathsf{dex}_{n_4}\}] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}, q_2\}, \langle s_7, s_8 \rangle, 1, \{t_2, t_6\}] \\
s_6 &\overset{def}{=} [n_6, \{\mathsf{den}_{n_6}\}, \{\mathsf{dex}_{n_6}\}] \\
s_7 &\overset{def}{=} [n_7, \{\mathsf{den}_{n_7}\}, \{\mathsf{dex}_{n_7}\}] \\
s_8 &\overset{def}{=} [n_8, \{\mathsf{den}_{n_8}\}, \{\mathsf{dex}_{n_8}\}]
\end{aligned}
$$

with transitions

$$
\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_5}, y, \perp) \\
t_2 &\overset{def}{=} (\mathsf{out}, \mathsf{true}, \mathsf{dex}_{n_8}, q_2, x, \perp) \\
t_3 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_5}, \mathsf{den}_{n_6}, y, \perp) \\
t_4 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_3}, \mathsf{den}_{n_2}, z, \perp) \\
t_5 &\overset{def}{=} (\mathsf{sib}, \mathsf{false}, q_1, \mathsf{den}_{n_3}, \perp, \perp) \\
t_6 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_7}, \mathsf{den}_{n_8}, x, \perp) \\
t_7 &\overset{def}{=} (\mathsf{out}, \mathsf{false}, q_2, q_1, \perp, \perp)
\end{aligned}
$$

In π_{klt}, it is as follows:

```
new $h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8$ in def {
  proc $Sn_1(enp) =$ def {
    proc $Sn_2(enp) =$ def {
      proc $Sn_4(enp) = y?.Sn_5(\mathsf{den}_{n_5});$
      proc $Sn_5(enp) =$ def {
        proc $Sn_7(enp) = x?.Sn_8(\mathsf{den}_{n_8});$
        proc $Sn_8(enp) =$
          $x?.(\mathsf{set}\, h_5 := r_8 \parallel r_8?enp.Sn_8(enp) \parallel Bq_2());$
        proc $Bq_2() = (\mathsf{set}\, h_2 := r_5 \parallel r_5?enp.Sn_5(enp) \parallel Bq_1())$
      } in
        if $enp =$ "$init$" then $Sn_7($"$init$"$)$
        else let $r =$ get $h_5$ in $r!($"$hist$"$);$
      proc $Bq_1() = Sn_3(\mathsf{den}_{n_3});$
      proc $Sn_6(enp) = \surd;$
    } in
      if $enp =$ "$init$" then $Sn_4($"$init$"$)$
      else let $r =$ get $h_2$ in $r!($"$hist$"$);$
    proc $Sn_3(enp) = z?.Sn_2(\mathsf{den}_{n_2});$
  } in $Sn_2($"$init$"$)$
} in $Sn_1($"$init$"$)$
```

Now define a map $\hat{\mathcal{T}}[\![\cdot]\!] : \mathsf{SM} \to \mathsf{KLT}$ in terms of a recursive map $\mathcal{T}_6[\![\cdot]\!] : \mathsf{SM} \to \mathsf{SM}_\top \to \mathsf{KLT}$, which takes as input the State Machine term, and its enclosing state, and returns the corresponding π_{klt} term. For simplicity and uniformity we give all states parameters $inp, acc, rej, enp, exit, exack, sh$. The map $\hat{\mathcal{T}}$ is intended to be applied at the top level, i.e. to a full State Machine, not enclosed within another State Machine. In this mapping we explicitly associate an index k with each state n_k so that the corresponding history cell is denoted h_k and reentry signal is r_k.

Definition 10. (Encoding history)

- Given $s \in \mathsf{SM}$, let $\hat{\mathcal{T}}[\![s]\!] \in \mathsf{KLT}$ be defined as:

$$\hat{\mathcal{T}}[\![s]\!] \overset{def}{=} \mathsf{new}\ \ h_1, ..., h_M, r_1, ..., r_M\ \mathsf{in}\ \mathcal{T}_6[\![s]\!]_\perp$$

  where M is the number of states, and where $\mathcal{T}_6$ is defined as follows:

- For a basic state $s \overset{def}{=} [n_k, A, B, en, ex]$ its translation is given by:

$$
\begin{array}{l}
\mathcal{T}_6[\![s]\!]_{s'} \stackrel{def}{=} \\
\quad \mathsf{proc}\ Sn_k(inp, acc, rej, enp, exit, exack, sh) = \\
\quad\quad inp?x. \\
\quad\quad\quad \mathsf{if}\ x = \text{``}x_1\text{''}\ \mathsf{then}\ acc!.Q_1 \\
\quad\quad\quad \mathsf{else\ if}\ x = \text{``}x_2\text{''}\ \mathsf{then}\ acc!.Q_2 \\
\quad\quad\quad \cdots \\
\quad\quad\quad \mathsf{else\ if}\ x = \text{``}x_m\text{''}\ \mathsf{then}\ acc!.Q_m \\
\quad\quad\quad \mathsf{else}\ rej!.Sn_k(inp, acc, rej, enp, exit, exack, sh) \\
\quad\quad +exit?.exack!
\end{array}
$$

where s's enclosing state is

$$s' = [n'_{k'}, A', B', S', d', T', en', ex']$$

and

$$T'' \stackrel{def}{=} \{t \in T' \mid \exists q \in B.\, q = \mathsf{src}(t)\}$$

is the set of transitions from T' whose source (q) is an exit point of state n; each

$$x_i \stackrel{def}{=} \mathsf{evt}(t_i)$$

is the trigger event of transition t_i in the set T''; and Q_i is the process that exists and goes to the target of the transition, defined as

$$
Q_i \stackrel{def}{=} \begin{cases}
L_{k,k'} \parallel Sn_i(inp, acc, rej, p_i, exit, exack, sh) & \text{if } \mathsf{kind}(t_i) = \mathsf{sib},\ \mathsf{trg}(t_i) = p_i, \\
 & \quad \exists s_i \in S'.\, p_i \in \mathsf{entries}(s_i), \\
 & \quad \text{and } n_i = \mathsf{name}(s_i) \\
L_{k,k'} \parallel Bq_i(sh) & \text{if } \mathsf{kind}(t_i) = \mathsf{out} \\
 & \quad \text{and } \mathsf{trg}(t_i) = q_i \in B'
\end{cases}
$$

where

$$
\begin{array}{rcl}
L_{k,k'} & \stackrel{def}{=} & \mathsf{set}\ h_{k'} := r_k \\
 & \parallel & r_k?(inp, acc, rej, p_i, exit, exack, sh).Sn_k(inp, acc, rej, p_i, exit, exack, sh)
\end{array}
$$

- For a composite state $s = [n, A, B, S, d, T, en, ex]$ (with enclosing state s'):

$$
\begin{array}{l}
\mathcal{T}_6[\![s]\!]_{s'} \stackrel{def}{=} \\
\quad \mathsf{proc}\ Sn_k(inp, acc, rej, enp, exit, exack, sh) = \\
\quad\quad \mathsf{def}\ \{D_1; ...; D_k; B_1; ...; B_l; C_{def}; H_{def}\}\ \mathsf{in} \\
\quad\quad\quad \mathsf{new}\ \ inp', acc', rej', exit', exack', sh'\ \mathsf{in} \\
\quad\quad\quad\quad (C(inp', acc', rej', enp, exit', exack', sh') \\
\quad\quad\quad\quad\ \parallel H(inp', acc', rej', exit', exack', sh'))
\end{array}
$$

where each D_i is $\mathcal{T}_6[\![s_i]\!]_s$ for each s_i in $S = s_{1..k}$; each B_i is a process definition for exit point $q_i \in B$, given by

$$B_i \overset{def}{=} \mathsf{proc}\ Bq_i(sh') = sh'! \parallel Q_j$$

where Q_j is the target of the exit point, and is defined as above, and C_{def} is the dispatcher defined as follows:

```
proc $C(inp', acc', rej', enp, exit', exack', sh') =$
    if        $enp = p_1$        then   $Sn_1(inp', acc', rej', p'_1, exit', exack', sh')$
    else if   $enp = p_2$        then   $Sn_2(inp', acc', rej', p'_2, exit', exack', sh')$
    ...
    else if   $enp = p_m$        then   $Sn_m(inp', acc', rej', p'_m, exit', exack', sh')$
    else if   $enp = \text{``}init''$   then   $Sn_d(inp', acc', rej', p'_d, exit', exack', sh')$
    else      let $r = \mathsf{get}h_k$ in $r!(inp', acc', rej', \text{``}hist'', exit', exack', sh')$
```

where each $p_i \in A$ is a named entry point of s connected to the entry point p'_i of a sub-state n_i via an incoming transition $t_i = (\mathsf{in}, \mathsf{false}, p_i, p'_i, \bot, \bot) \in T$. Here we assume that the default state is n_d, with the initial transition connected to the entry point p'_d. Finally, H_{def} is the definition of the event handler H, as follows:

```
proc $H(inp', acc', rej', exit', exack', sh') =$
    $inp?x.inp'!x.$
        $(acc'?.acc!.H(inp', acc', rej', exit', exack', sh')$
        $+rej'?.$
            if $x = \text{``}x_1\text{''}$ then $exit'!.exack'?.acc!.Q_1$
            else if $x = \text{``}x_2\text{''}$ then $exit'!.exack'?.acc!.Q_2$
            ...
            else if $x = \text{``}x_m\text{''}$ then $exit'!.exack'?.acc!.Q_m$
            else $rej!.H(inp', acc', rej', exit', exack', sh'))$
    $+exit?.exit'!.exack'?.exack!$
    $+sh'?.\surd$
```

where T'' and Q_i are defined as for basic states.

4.8 Adding actions

So far we have not dealt with actions. There are two main issues to be addressed in order to support actions: first, how are individual actions encoded in π_{klt} and second, where should the be executed.

To address the first question, we considered an existing set of actions $\mathcal{A}$ without specifying what exactly are these actions. Normally these actions would be given in some action language. However, the order of execution (the second issue) is independent of such action language, and therefore it is useful to keep this set abstract, and assume that we have a translation $\alpha : \mathcal{A}_\perp \rightarrow \mathsf{KLT}$ which maps each action to the corresponding π_{klt} term.

Once we assume the action translation, we can focus on where to put the resulting translations. We have three kinds of action: entry actions, exit actions and transition actions. Entry actions must be executed whenever we enter a state. Similarly for exit actions. Transition actions are executed whenever the transition is taking place, after exiting the source state and before entering the target state.[13] This means that the process Sn for a state $[n, ..., en, ex]$ must begin by executing $\alpha(en)$ and that $\alpha(ex)$ must be executed when leaving the state, this is in the process Bq for each exit point q.

Example 14. Let us extend Example 11 with actions as follows:

$$\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_3\rangle, 1, \{t_1\}, en_1, ex_1] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}\}, \{\mathsf{dex}_{n_2}\}, \langle s_4, s_5\rangle, 1, \{t_2\}, en_2, ex_2] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}, en_3, ex_3] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}\}, \{\mathsf{dex}_{n_4}\}, \langle s_6, s_7\rangle, 1, \{t_3\}, en_4, ex_4] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}\}, en_5, ex_5] \\
s_6 &\overset{def}{=} [n_6, \{\mathsf{den}_{n_6}\}, \{\mathsf{dex}_{n_6}\}, en_6, ex_6] \\
s_7 &\overset{def}{=} [n_7, \{\mathsf{den}_{n_7}\}, \{\mathsf{dex}_{n_7}\}, en_7, ex_7]
\end{aligned}$$

with transitions

$$\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_2}, \mathsf{den}_{n_3}, x, a_1) \\
t_2 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_4}, \mathsf{den}_{n_5}, y, a_2) \\
t_3 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_6}, \mathsf{den}_{n_7}, z, a_3)
\end{aligned}$$

In π_{klt} this would be represented as:

[13]There are some exceptions to this rule, namely the so-called *internal transitions*, but we do not address these at this point.

```
proc $Sn_1(enp) = \{$
  proc $Sn_2(enp) = \{$
    proc $Sn_4(enp, exit, exack) = \{$
      proc $Sn_6(enp, exit, exack) =$
        $\alpha(en_6); (z?.\alpha(ex_6); \alpha(a_3); Sn_7(\mathsf{den}_{n_7}, exit, exack) + exit?.\alpha(ex_6); exack!);$
      proc $Sn_7(enp, exit, exack) = \alpha(en_7); exit?.\alpha(ex_7); exack!;$
      proc $H(exit', exack') =$
        $y?.exit'!.exack'?.\alpha(ex_4); \alpha(a_2); Sn_5(\mathsf{den}_{n_5}, exit, exack)$
        $+exit?.exit'!.exack'?.\alpha(ex_4); exack!$
    $\}$ in new $exit', exack'$ in
      $\alpha(en_4); (Sn_6(\mathsf{den}_{n_6}, exit', exack') \parallel H(exit', exack'));$
    proc $Sn_5(enp, exit, exack) = \alpha(en_5); exit?.\alpha(ex_5); exack!;$
    proc $H(exit', exack') = x?.exit'!.exack'?.\alpha(ex_2); \alpha(a_1); Sn_3(\mathsf{den}_{n_3})$
  $\}$ in new $exit', exack'$ in $\alpha(en_1); (Sn_4(\mathsf{den}_{n_4}, exit', exack') \parallel H(exit', exack'));$
  proc $Sn_3(enp) = \surd;$
$\}$ in $Sn_3(\mathsf{den}_{n_3})$
```

Suppose for example that the machine is in state n_6 (and therefore in states n_4 and n_2 as well), and then an event y arrives.

Now define a map $\hat{\mathcal{T}}[\![\cdot]\!] : \mathsf{SM} \to \mathsf{KLT}$ in terms of a recursive map $\mathcal{T}_7[\![\cdot]\!] : \mathsf{SM} \to \mathsf{SM}_\top \to \mathsf{KLT}$, which takes as input the State Machine term, and its enclosing state, and returns the corresponding π_{klt} term. For simplicity and uniformity we give all states parameters $inp, acc, rej, enp, exit, exack, sh$. The map $\hat{\mathcal{T}}$ is intended to be applied at the top level, i.e. to a full State Machine, not enclosed within another State Machine. In this mapping we explicitly associate an index k with each state n_k so that the corresponding history cell is denoted h_k and reentry signal is r_k. The map also depends on a translation $\alpha : \mathcal{A}_\perp \to \mathsf{KLT}$ from the set of actions to the set of π_{klt}-terms.

Definition 11. (Encoding actions)

- Given $s \in \mathsf{SM}$, let $\hat{\mathcal{T}}[\![s]\!] \in \mathsf{KLT}$ be defined as:

$$\hat{\mathcal{T}}[\![s]\!] \overset{def}{=} \text{new } h_1, ..., h_M, r_1, ..., r_M \text{ in } \mathcal{T}_7[\![s]\!]_\perp$$

  where M is the number of states, and where $\mathcal{T}_7$ is defined as follows:

- For a basic state $s \overset{def}{=} [n_k, A, B, en, ex]$ its translation is given by:[14]

[14]This definition adds an auxiliary internal definition $K()$. This is because, if the state cannot handle a transition it must trigger a reject signal ($rej!$) and go back to waiting, without executing the entry action again.

```
$\mathcal{T}_7[\![s]\!]_{s'} \overset{def}{=}$
  proc $Sn_k(inp, acc, rej, enp, exit, exack, sh)$ = def {
    proc $K()$ =
      $inp?x.$
        if $x =$ “$x_1$” then $acc!.Q_1$
        else if $x =$ “$x_2$” then $acc!.Q_2$
        $\ldots$
        else if $x =$ “$x_m$” then $acc!.Q_m$
        else $rej!.K()$
      $+exit?.\alpha(ex); exack!$
    } in $\alpha(en); K()$
```

where s’s enclosing state is

$$s' = [n'_{k'}, A', B', S', d', T', en', ex']$$

and

$$T'' \overset{def}{=} \{t \in T' \mid \exists q \in B.\, q = \mathsf{src}(t)\}$$

is the set of transitions from T' whose source (q) is an exit point of state n; each

$$x_i \overset{def}{=} \mathsf{evt}(t_i)$$

is the trigger event of transition t_i in the set T''; and Q_i is the process that exists and goes to the target of the transition, defined as

$$Q_i \overset{def}{=} \begin{cases} L_{k,k'} \parallel E_i; Sn_i(inp, acc, rej, p_i, exit, exack, sh) & \text{if } \mathsf{kind}(t_i) = \mathsf{sib},\ \mathsf{trg}(t_i) = p_i, \\ & \quad \exists s_i \in S'.\, p_i \in \mathsf{entries}(s_i), \\ & \quad \text{and } n_i = \mathsf{name}(s_i) \\ L_{k,k'} \parallel E_i; Bq_i(sh) & \text{if } \mathsf{kind}(t_i) = \mathsf{out} \\ & \quad \text{and } \mathsf{trg}(t_i) = q_i \in B' \end{cases}$$

where

$$\begin{aligned} L_{k,k'} &\overset{def}{=} \mathsf{set}\, h_{k'} := r_k \\ &\parallel\ r_k?(inp, acc, rej, p_i, exit, exack, sh).Sn_k(inp, acc, rej, p_i, exit, exack, sh) \\ E_i &\overset{def}{=} \alpha(ex); \alpha(\mathsf{act}(t_i)) \end{aligned}$$

- For a composite state $s = [n, A, B, S, d, T, en, ex]$ (with enclosing state s'):

```
$\mathcal{T}_7[\![s]\!]_{s'} \overset{def}{=}$
  proc $Sn_k(inp, acc, rej, enp, exit, exack, sh)$ =
    def $\{D_1; ...; D_k; B_1; ...; B_l; C_{def}; H_{def}\}$ in
      new $inp', acc', rej', exit', exack', sh'$ in
        $\alpha(en)$;
        $(C(inp', acc', rej', enp, exit', exack', sh')$
          $\parallel H(inp', acc', rej', exit', exack', sh'))$
```

where each D_i is $\mathcal{T}_7[\![s_i]\!]_s$ for each s_i in $S = s_{1..k}$; each B_i is a process definition for exit point $q_i \in B$, given by

$$B_i \overset{def}{=} \mathsf{proc}\ Bq_i(sh') = sh'! \parallel Q_j$$

where Q_j is the target of the exit point, and is defined as above, and C_{def} is the dispatcher defined as follows:

```
proc $C(inp', acc', rej', enp, exit', exack', sh') =$
    if        $enp = p_1$          then  $\alpha(\mathsf{act}(t_1)); Sn_1(inp', acc', rej', p'_1, exit', exack', sh')$
    else if   $enp = p_2$          then  $\alpha(\mathsf{act}(t_2)); Sn_2(inp', acc', rej', p'_2, exit', exack', sh')$
    $\ldots$
    else if   $enp = p_l$          then  $\alpha(\mathsf{act}(t_l)); Sn_l(inp', acc', rej', p'_l, exit', exack', sh')$
    else if   $enp = \text{“}init''$  then  $\alpha(\mathsf{act}(t_d)); Sn_d(inp', acc', rej', p'_d, exit', exack', sh')$
    else      let $r = \mathsf{get}h_k$ in $r!(inp', acc', rej', \text{“}hist'', exit', exack', sh')$
```

where each $p_i \in A$ is a named entry point of s connected to the entry point p'_i of a sub-state n_i via an incoming transition $t_i = (\mathsf{in}, \mathsf{false}, p_i, p'_i, \bot, \bot) \in T$. Here we assume that the default state is n_d, with the initial transition connected to the entry point p'_d. Finally, H_{def} is the definition of the event handler H, as follows:

```
proc $H(inp', acc', rej', exit', exack', sh') =$
    $inp?x.inp'!x.$
        $(acc'?.acc!.H(inp', acc', rej', exit', exack', sh')$
        $+rej'?.$
            if $x = \text{“}x_1\text{”}$ then $exit'!.exack'?.acc!.Q_1$
            else if $x = \text{“}x_2\text{”}$ then $exit'!.exack'?.acc!.Q_2$
            $\ldots$
            else if $x = \text{“}x_m\text{”}$ then $exit'!.exack'?.acc!.Q_m$
            else $rej!.H(inp', acc', rej', exit', exack', sh'))$
    $+exit?.exit'!.exack'?.\alpha(ex); exack!$
    $+sh'?.\surd$
```

where T'' and Q_i are defined as for basic states.

5 Concluding remarks

We have presented a map from UML-RT State Machines into the kiltera language which preserves the machine's hierarchical structure. By preserving this structure we can easily recognize elements of the original machine in the encoded representation. Furthermore, the encoding's modular structure includes specific components (the handler and the dispatcher) which address the particular semantics of UML-RT State Machines. This results in a separation of concerns where alternative semantics and policies could be put in place with minimal effect on the rest of the structure.

Our goal is to be comprehensive in the treatment of UML-RT semantics. Nevertheless, the present mapping still lacks support for some features, in particular choice and junction points, and internal transitions.

This work is part of our effort to give a formal semantics and develop analysis tools and techniques for UML-RT as a whole, not only State Machines. Therefore as part of this effort, we are also working towards encoding structure diagrams, including capsules and objects in general, in kiltera. The mapping provided here will be used in such encoding.

References

[1] *IBM Rational Rose Technical Developer, Version 7.0.* `www-01.ibm.com/software/awdtools/developer/technical`.

[2] *IBM Rational Software Architect, RealTime Edition, Version 7.5.2.* `publib.boulder.ibm.com/infocenter/rsarthlp/v7r5m1`.

[3] Unified Modeling Language (UML), UML 2.2 Spec.

[4] D. R. Jefferson. Virtual Time. *ACM-TOPLAS*, 7(3):404–425, July 1985.

[5] E. Posse. A real-time extension to the π-calculus. Tech. Report 2009-557, School of Computing – Queen's University, http://www.cs.queensu.ca, 2009.

[6] E. Posse and J. Dingel. Theory and implementation of a real-time extension to the π-calculus. In *Proc. Int. Conf. on Formal Techniques for Distributed Systems (FMOODS&FORTE'10)*, LNCS, 2010. Accepted.

[7] B. Selic. Using UML for modeling complex real time systems. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *LNCS*, 1998.

[8] B. Selic, G. Gullekson, and P.T. Ward. *Real-time Object Oriented Modeling and Design.* J. Wiley & Sons, 1994.

[9] M. von der Beeck. A structured operational semantics for UML-statecharts. *SoSyM*, 1(2):130–141, 2002.

[10] B.P. Zeigler, H. Praehofer, and T.G. Kim. *Theory of Modeling and Simulation (2nd ed.).* Academic Press, 2000.