Mapping UML State Machines to kiltera

Technical Report 2008-552

Ernesto Posse Juergen Dingel
Applied Formal Methods Group
Software Technology Lab
Queen's University
Kingston, Ontario, Canada

January 5, 2009

Abstract

As part of an effort to clarify the semantics of the UML we investigate the semantics of UML State Machines and in particular we present an encoding of a significant subset of UML State Machines into a process algebra named kiltera, closely related to the π-calculus. Our approach differs from other related attempts in that it provides a precise, executable, compositional, and extensible semantics of UML State Machines in terms of a precise language, which furthermore does not resort to flattening the state-space, and thus avoids the state-space explosion problem. This approach can serve as the basis for tool support for UML State Machine models, in particular, simulators, code-generators, debuggers and model-checkers. We compare our encoding with several other proposed approaches.

Contents

1 Introduction

The Unified Modeling Language (UML, [20]) has become the *de facto* standard for software modelling, and has received considerable attention from software practitioners, tool developers, as well as researchers in Software Engineering.

Despite its growing adoption, the UML lacks a formal semantics. This is major source of problems, as different tool developers and vendors produce software modelling environments which are claimed to be "UML compliant" but may, in fact, disagree in terms of the behaviours produced by generated systems. Furthermore, this lack of formal semantics also hinders the analyzability of models and, as a result, there is a lack of analysis tools for UML such as model-checkers.

The need for a formal foundation for the UML is exemplified by the increased interest in *executable* UML. The Object Management Group has issued a Request For Proposal (RFP) for a semantics for an executable UML foundation [18]. This Request For Proposal has been answered in [19], but the resulting semantics is defined in terms of a Java-like language, making the semantics not as abstract as possible and too implementation-dependent in the sense that it relies too heavily on the (non-trivial) semantics of the Java-like language used, which in turn compromises analyzability without solving all ambiguities. For this reason it is desirable to search for a formal semantics which is based on some simple, well-defined formalism, thus clarifying ambiguities and enabling model analysis.

Formalizing the UML is a non-trivial task. The UML specification [20] is a very large document full of technical details, and with a significant amount of ambiguities. Many of these ambiguities are deliberate, providing *semantic variation points*, in order to accommodate different tool developers. But others are the result of lack of specification and clarity. Nevertheless, the problem of formalizing its semantics can be broken down, as the UML consists of several sub-languages, each of which can be studied and understood with a certain degree of independence from the others. The UML sub-languages are broadly divided into two categories: structural and behavioural. Structural sub-languages such as Class Diagrams, describe the structure of a software system. Behavioural sub-languages describe the behaviour of systems. Since the focus of the UML is modelling Object-Oriented software, behavioural diagrams describe the behaviour of objects. There are three main kinds of behavioural diagrams: Activity Diagrams, Interaction Diagrams and State Machines. This report is concerned with defining the semantics of State Machines.

We propose a formal semantics for a significant subset of UML State Machines (which we also call statecharts) by mapping them into a process algebra called kiltera [22, 23] closely related to the π-calculus [15, 14]. Furthermore, we propose a textual syntax for state machine diagrams, based on the syntax defined in [28].

A question that may arise is why did we choose kiltera as a target language instead of better known languages such as CSP [9, 25], ASMs [2] or even the π-calculus [15, 14] upon which kiltera is based. This selection came down to the choice of operators provided by the language as well as the facilities to simulate the resulting models. In particular, kiltera provides some higher-level constructs such as pattern-matching which facilitate the description of the generated models and render them more readable. Furthermore, UML State Machines allow time-triggered transitions, but the existing timed variants of the aforementioned calculi do not easily capture such semantics.

The mapping described in this report is based on [4, 3]. This previous work proposes an informal transformation from classical (STATEMATE) statecharts [8] into the DEVS formalism [32].

Our mapping has several characteristics that make it appealing: it provides a precise, formal semantics in a compositional, non-flattening fashion which is executable and extensible enough to support several semantic variation points. The benefits of being a compositional mapping are multiple:

- the semantics of a state machine is uniquely determined by the semantics of its component sub-states, thus supporting compositional reasoning about models, as well as component *replaceability* (i.e., a sub-state machine can be replaced by an equivalent one without changing the behaviour of the whole state machine),

- it does not rely on flattening the statechart, thus resulting in a more compact representation,
- the structure of the statechart is mimicked by the resulting kiltera model, thus each state is in a one-to-one correspondence with a component in the kiltera model, which facilitates analysis, traceability, and tool support (e.g. debuggers, animation, etc).

We concentrate on a subset of State Machines, rather than dealing with the full specification in all detail in order to simplify the treatment. We focus on the following features of the official UML State Machine specification:

- Composite states, including both or-states (also known as sequential states) and and-states (also known as concurrent states),
- Inter-level transitions (i.e. transitions "crossing boundaries")
- Group transitions
- Entry and exit actions
- Transition actions

We do not deal in the present version of the mapping with the following features of the official UML State Machine specification:

- History states
- Pseudo states, in particular, forks, joins, condition points, junctions, entry points and exit points.
- Transition guards

Nevertheless, our mapping is extensible enough to be able to deal with all features of the official UML specification.

1.1 Related work

There have been multiple efforts to formalize the semantics of UML State Machines, as well as other variants of statecharts. We now discuss some of these approaches.

Yeung et al (CSP semantics) One approach is presented in [29] where a subset of UML State Machines is mapped into CSP [9, 24, 25]. This approach is comparable to ours in that the target of the mapping is a process algebra (CSP) and they use a notion of paths for transitions similar to ours, but that's where the similarities end. The most striking difference is that they map a statechart to a flattened state machine in CSP: the hierarchical structure of the state machine is lost and each group transition[1] is encoded in each possible sub-state of its source. Furthermore this mapping does not enforce priorities between conflicting transitions at different levels of nesting, as required by the UML specification. Like ours, they do not deal with History states, other pseudo-states, or transition guards, but unlike ours, it is not very clear how their mapping would have to be extended to deal with such features.

This approach seems to build on the mapping introduced in [17], although this earlier work also deals with choice pseudo-states (conditionals) and it presents a prototype translation tool from Rational Rose State Diagrams to CSP code suitable as input for FDR [6] for verification.

Von der Beeck (SOS semantics) In [28] an alternative approach is presented, in which a textual syntax for UML State Machines is introduced and an operational semantics is defined in the style of Plotkin's Structural Operational Semantics [21] as a set of inference rules defining a labelled-transition system for statecharts. This approach has the advantage of being compositional and it even deals with features such as shallow and deep history. Nevertheless it is not a directly executable semantics, but rather, it provides the specification for a simulation engine or interpreter. This contrasts with our approach of mapping statecharts to another language for which we have a simulator. A major drawback of this approach compared to ours is that integrating timed transitions is non-trivial, and would involve modifying and extending the inference rules and possibly the syntax, whereas in our approach this extension can be dealt with by using the timing constructs of our target language.

Van Langenhove (EHA/Kripke/SMV semantics) Another approach is presented in [26], where State Machines are represented as Extended Hierarchical Automata, a kind of automata where each state may be associated with another automaton. These hierarchical automata are given an operational semantics as Kripke structures which are then mapped to SMV [13].

The mapping presented there has the advantage of being non-flattening, but it is not clear whether it is compositional or not, in the sense that it is not clear whether the final SMV target code of a statechart can be seen as the combination of the translations of the component sub-statecharts. Furthermore, their mapping already requires the definition of an operational semantics for the Extended Hierarchical Automata, which is given as a Kripke structure whose states, called *configurations*, already carry a lot of the machinery required of UML State Machines, such as an event queue and a history. Furthermore, non-compliant restrictions such as a maximum event-queue size, are imposed on the target code. But defining the semantics of State Machines in terms of something that already has those concepts embedded as primitive hardly clarifies the semantics. A semantics should define something more complex in terms of something simpler. This contrasts with our approach, where we do not

[1] A *group transition* is a transition whose source is a composite state.

impose extraneous, target language restrictions, and rely on a language which does not include concepts of history or event-queues as primitive.

Lam and Padget (π-calculus semantics) An approach which maps State Machines into the π-calculus, the process algebra upon which kiltera is based, was introduced in [11] and further developed in [12] and [10].

Their approach associates states with π-calculus terms, and a protocol of channel exchanges is used to model reception and handling of events. This approach supports both shallow and deep history. The thesis [10] also presents a tool that generates input code for the Mobility Workbench (MWB [27]) that provides deadlock detection and equivalence checking between the generated π-calculus code from two statecharts (open bisimilarity) and also generates input code for the NuSMV model-checker [5].

One of the main weaknesses of their approach is that the encoding does not clearly describe how the hierarchical structure of a statechart is represented, and in particular it is not clear how the encoding would accommodate an arbitrarily deep hierarchy. Although the authors claim that their scheme respects the lowest-first firing priority, the encoding and the examples provided seems to support only one level of nesting between states. This puts into question the claim of compositionality of this approach. There is no mention of how to deal with inter-level and group transitions either, all fundamental features in UML State Machines.

By relying on the pure π-calculus rather than a higher-level language, their mapping makes use of complicated encodings for simple activities such as evaluation of a guard. Furthermore their mapping makes use of the unrestricted choice operator $+$ which is very difficult to implement in practice (see [16]).

A controversial aspect of this translation is that the State Machines modelled also include Activities, thus describing a hybrid formalism between UML State Machines and UML Activity Diagrams. This addition seems to introduce confusion in the semantics rather than clarify it. Nevertheless, it appears to be a relatively orthogonal issue which could be taken away.

Börger et al (ASMs semantics) Perhaps the most comprehensive approach is that of [1] where the dynamics of UML State Machines are described using Abstract State Machines [2]. This approach takes into account both sequential states (or-states), concurrent states (and-states) as well as history pseudo-states, and other features such as deferred events.

In this approach, the structure of the UML State Machine is encoded as part of the state in the ASM, together with additional machinery used to keep track of the current states, history, etc. Executing the state machine is performed by ASM agents which choose among enabled transitions and execute the selected transition by removing states which are exited and inserting entered states in the table that keeps track of the current configuration.

This approach is different to others in that rather than mapping UML State Machines to a "program" in the target language, they are mapped to a data structure in the target language, and a general algorithm is implemented in the target language which executes (interprets) this data structure.

A drawback of this approach is that, while model-checking techniques exist for ASMs (e.g. [7]), using these techniques on the approach presented would allow the verification of properties of the simulation algorithm itself, rather than properties of a given statechart. Similarly, taking the "interpreter" approach to semantics makes the comparison of statecharts more difficult: given two state machines to compare, one has to consider the steps that the interpreter goes through, rather than the steps that a semantic representation of the state machines would follow. Furthermore, one wants to understand the behaviour of a state machine which does not have certain features (e.g. history states), a mapping to some language would not encode the corresponding features and therefore the meaning associated to the state machine does not have elements which do not affect its behaviour. By contrast, in a semantics approach based on an interpreter, such as this ASM approach, when looking at the meanning of a state machine one has to consider the interpreter and how it deals with all features in the formalism.

Another drawback of this approach is that, quoting [1]: "The UML requirement that an object is not allowed to remain in a pseudostate, but has to immediately move to a normal state, cannot be guaranteed by the rules themselves, but has to be imposed as an integrity constraint on the permissible runs." In other words, the algorithm itself is not sufficient to emulate the precise semantics of statecharts and one must resort to an extraneous constraint on the possible executions, limiting the direct executability of the semantics.

Borland and Vangheluwe (DEVS semantics) The mapping upon which our work is based was introduced in [4, 3]. That work presents an *informal* translation from STATEMATE statecharts into DEVS [32, 30, 31] models.

Because of the significant differences between STATEMATE statecharts and UML statecharts, as well as the differences between kiltera and DEVS, our mapping departs significantly from the former in many respects, but the compositionality of the approach is the same, including the idea of relay processes to handle events within a composite state, as well as the routing mechanism within the hierarchical structure of the statechart.

One of the main differences in their approach, stemming from the STATEMATE semantics is that incoming events are treated in a highest-first firing priority, as opposed the UML State Machines. Furthermore their approach does not guarantee run-to-completion semantics.

The most important difference, however, is that their work constitutes an informal description of the dynamics of statecharts, whereas our approach presents a precise formal semantics.

1.2 Organization of this report

The remainder of this report is organized as follows: in Section 2 we provide some preliminary definitions and notation used throughout the report. In particular we introduce a new textual syntax for State Machines in Section 2.2 and present an informal account of the kiltera language in Section 2.3. Section 3 presents the mapping itself. We begin describing the structure of the model produced by the mapping (Subsection 3.1), followed by a description of the messages that flow between components in the generated model (Subsection 3.2) and actions (Subsection 3.3). Then we describe the mapping of basic states (Subsection 3.4), or-states (Subsection 3.5) and and-states (Subsection 3.6).

2 Preliminaries

2.1 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

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

Prepending an item x to a sequence $\vec{a} = \langle a_1, ..., a_k\rangle$ is denoted $x\vec{a}$, so $x\vec{a} \overset{def}{=} \langle x\rangle \cdot \vec{a} = \langle x, a_1, ..., a_k\rangle$. We define $\text{last}(\langle a_1, ..., a_k\rangle) \overset{def}{=} a_k$. We denote $\text{rev}(\vec{a})$ for the reverse of the sequence, i.e. $\text{rev}(\langle a_1, ..., a_k\rangle) \overset{def}{=} \langle a_k, ..., a_1\rangle$. We write $|\vec{a}|$ for the length of the sequence $\vec{a}$.

A sequence defines a total order on its elements according to their positions: let $\vec{a} = \langle a_1, ..., a_k\rangle$ be some sequence, then we write $a_i \preceq a_j$ if $i \leq j$.

We will also use *sequence comprehension notation*: if I is some totally ordered set (possibly a sequence), $\varphi(x)$ is an expression with free variable x, and $\psi(x)$ is some predicate on x, then the expression $\langle \varphi(x) \,|\, x \in I,\, \psi(x)\rangle$, also written $\langle \varphi(x) \,|\, \psi(x)\rangle_{x\in I}$ denotes the sequence of all $\varphi(x)$ such that $\psi(x)$ holds, preserving the order over I, this is, if $x \leq x'$ for some $x, x' \in I$ then $\varphi(x) \preceq \varphi(x')$. For example, $\langle x^2\rangle_{x\in\{1,2,3\}} = \langle 1, 4, 9\rangle$.

Definition 1. (Prefix) Let $\vec{a}$ and $\vec{b}$ be a pair of sequences. We say that $\vec{a}$ is ***a prefix*** of $\vec{b}$, written $\vec{a} \sqsubseteq \vec{b}$ if there is a sequence $\vec{w}$ such that $\vec{a} \cdot \vec{w} = \vec{b}$.

Remark 1. $\sqsubseteq$ is a partial order, this is, it is reflexive, transitive and anti-symmetric.

Definition 2. (Prefix removal and common prefix) Let $\vec{a}$ be a sequence and $\vec{b}$ some prefix of $\vec{a}$, i.e. $\vec{a} = \vec{b} \cdot \vec{w}$ for some $\vec{w}$. Then, we write $\vec{a} - \vec{b}$ for $\vec{w}$.

This can be defined recursively as follows:

$$\begin{array}{rcll} \epsilon - x\vec{b} & \overset{def}{=} & \epsilon & \\ x\vec{a} - \epsilon & \overset{def}{=} & x\vec{a} & \\ x\vec{a} - x\vec{b} & \overset{def}{=} & \vec{a} - \vec{b} & \\ x\vec{a} - y\vec{b} & \overset{def}{=} & x\vec{a} & \text{if } x \neq y \end{array}$$

Given two sequences $\vec{a}$ and $\vec{b}$, $\vec{a} \sqcap \vec{b}$ denotes ***the greatest common prefix*** of $\vec{a}$ and $\vec{b}$, i.e. $\vec{a} \sqcap \vec{b} \sqsubseteq \vec{a}$, $\vec{a} \sqcap \vec{b} \sqsubseteq \vec{b}$ and for any prefix $\vec{u}$ of both $\vec{a}$ and $\vec{b}$, $\vec{u} \sqsubseteq \vec{a} \sqcap \vec{b}$. This can be defined recursively as follows:

$$\begin{array}{rcll} \epsilon \sqcap \vec{b} & \overset{def}{=} & \epsilon & \\ \vec{a} \sqcap \epsilon & \overset{def}{=} & \epsilon & \\ x\vec{a} \sqcap y\vec{b} & \overset{def}{=} & \epsilon & \text{if } x \neq y \\ x\vec{a} \sqcap x\vec{b} & \overset{def}{=} & x(\vec{a} \sqcap \vec{b}) & \end{array}$$

Remark 2. The greatest common prefix is nothing but the greatest lower bound according the the prefix partial order $\sqsubseteq$.

2.2 Statechart syntax

Let $\mathcal{N}_S, \mathcal{N}_T, \Pi, \mathcal{A}$ be the sets of all possible state names, transition names, events and actions respectively. We use $n, m, ...$ for state names in $\mathcal{N}_S$ and $\underline{t}_1, \underline{t}_2, ...$ for transition names in $\mathcal{N}_T$. We write $e_1, e_2, ...$ for events in Π and $a_1, a_2, ...$ for actions in $\mathcal{A}$. We assume that each state is labelled with a unique name.

Definition 3. (Statechart terms) The set SC of statechart terms is defined according to the following BNF where $n \in \mathcal{N}_S$ and $s, s_1, ..., s_k$ range over SC :

$$\begin{array}{rcll} s & ::= & [n, (en, ex)] & \text{Basic-state} \\ & | & [n, (s_1, ..., s_k), T, (en, ex)] & \text{Or-state} \\ & | & [n, (s_1, ..., s_k), (en, ex)] & \text{And-state} \end{array}$$

Here $en, ex \in \mathcal{A} \cup \{\bot\}$ and $T \subseteq \mathsf{TR}$ where $\mathsf{TR} \overset{def}{=} \mathcal{N}_T \times \mathcal{N}_S^* \times \Pi \times \mathcal{A} \times \mathcal{N}_S^*$ is the set of transitions of an or-state, subject to the condition that if $s = [n, (s_1, ..., s_k), T, (en, ex)]$ then for each $t \in T$ with $t = (\underline{t}, \vec{o}, e, a, \vec{d})$, $\vec{o} = \langle o_1, ..., o_k \rangle$ and $\vec{d} = \langle d_1, ..., d_l \rangle$, the following holds[2]:

1. $o_i = \mathsf{parent}_s(o_{i+1})$ for each $i \in \{1, ..., k-1\}$,

2. $d_i = \mathsf{parent}_s(d_{i+1})$ for each $i \in \{1, ..., l-1\}$,

[2]Note that conditions 3 and 4 state that the the sequences for source ($\vec{o}$) and target ($\vec{d}$) of the transition begin with the state name of the state that contains the transition, rather than from the "root" of the state machine; in particular, the state containing the transition is the least common ancestor of the source and target of the transition.

3. $n = \mathsf{parent}_s(o_1)$, and

4. $n = \mathsf{parent}_s(d_1)$

where the function $\mathsf{parent}_s : \mathcal{N}_S \to \mathcal{N}_S$ which gives the name of the enclosing state (a.k.a. the parent) of a state with a given name within the statechart s, is defined as follows for each non-basic statechart term s:

$$\begin{array}{rcll}
\mathsf{parent}_{[n,s_{1..k},T,(en,ex)]}(m) & \overset{def}{=} & n & \text{if } \exists i \in \{1,...,k\}.\, \mathsf{name}(s_i) = m \\
\mathsf{parent}_{[n,s_{1..k},T,(en,ex)]}(m) & \overset{def}{=} & \mathsf{parent}_{s_j}(m) & \text{if } \neg\exists i \in \{1,...,k\}.\, \mathsf{name}(s_i) = m \\
 & & & \quad \wedge\, m \in \mathsf{descendants}(s_j) \\
\mathsf{parent}_{[n,s_{1..k},(en,ex)]}(m) & \overset{def}{=} & n & \text{if } \exists i \in \{1,...,k\}.\, \mathsf{name}(s_i) = m \\
\mathsf{parent}_{[n,s_{1..k},(en,ex)]}(m) & \overset{def}{=} & \mathsf{parent}_{s_j}(m) & \text{if } \neg\exists i \in \{1,...,k\}.\, \mathsf{name}(s_i) = m \\
 & & & \quad \wedge\, m \in \mathsf{descendants}(s_j)
\end{array}$$

with the functions $\mathsf{name} : \mathsf{SC} \to \mathcal{N}_S$ and $\mathsf{descendants} : \mathsf{SC} \to 2^{\mathcal{N}_S}$ given by:

$$\begin{array}{rcl}
\mathsf{name}([n,(en,ex)]) & \overset{def}{=} & n \\
\mathsf{name}([n,s_{1..k},T,(en,ex)]) & \overset{def}{=} & n \\
\mathsf{name}([n,s_{1..k},(en,ex)]) & \overset{def}{=} & n
\end{array}$$

and

$$\begin{array}{rcl}
\mathsf{descendants}([n,(en,ex)]) & \overset{def}{=} & \emptyset \\
\mathsf{descendants}([n,s_{1..k},T,(en,ex)]) & \overset{def}{=} & \{\mathsf{name}(s_i)\}_{i\in 1..k} \cup \bigcup_{i \in 1..k} \mathsf{descendants}(s_i)
\end{array}$$

Given an or-state $s = [n,(s_1,...,s_k),T,(en,ex)]$, we call the first sub-state s_1 the ***default state*** of s.

Given a transition $t = (\underline{t},\vec{o},e,a,\vec{d})$ we define $\mathsf{name}(t) \overset{def}{=} \underline{t}$ as the name of the transition, $\mathsf{qsrc}(t) \overset{def}{=} \vec{o}$ as the qualified source of t, $\mathsf{src}(t) \overset{def}{=} \text{last}(\mathsf{qsrc}(t))$ as the source of t, $\mathsf{evt}(t) \overset{def}{=} e$ as the trigger event of t, $\mathsf{act}(t) \overset{def}{=} a$ as the action of t, $\mathsf{qtrg}(t) \overset{def}{=} \vec{d}$ as the qualified target of t, and $\mathsf{trg}(t) \overset{def}{=} \text{last}(\mathsf{qtrg}(t))$ as the target of t.

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

Example 1. Consider the statechart depicted in Figure 1. This statechart contains only or-states. In our syntax this statechart is described by the term s_1 where:

$$\begin{array}{rcl}
s_1 & \overset{def}{=} & [n_1,(s_2,s_3),\{t_1,t_2\}] \\
s_2 & \overset{def}{=} & [n_2,(s_4,s_5),\{t_3\}] \\
s_3 & \overset{def}{=} & [n_3] \\
s_4 & \overset{def}{=} & [n_4] \\
s_5 & \overset{def}{=} & [n_5]
\end{array}$$

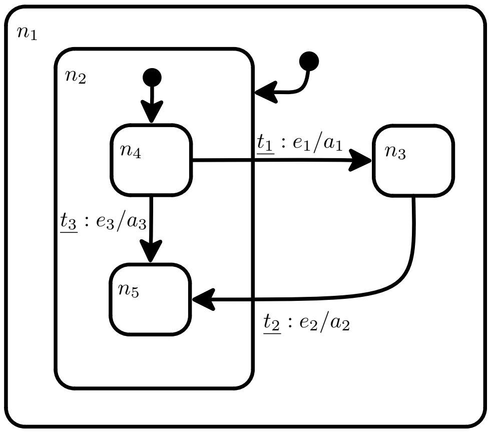
n_1
n_2
n_4
n_3
n_5
$\underline{t_1} : e_1/a_1$
$\underline{t_3} : e_3/a_3$
$\underline{t_2} : e_2/a_2$

Figure 1: A simple statechart.

where

$$\begin{aligned}
t_1 &\overset{def}{=} (\underline{t}_1, \langle n_2, n_4\rangle, e_1, a_1, \langle n_3\rangle) \\
t_2 &\overset{def}{=} (\underline{t}_2, \langle n_3\rangle, e_2, a_2, \langle n_2, n_5\rangle) \\
t_3 &\overset{def}{=} (\underline{t}_3, \langle n_4\rangle, e_3, a_3, \langle n_5\rangle)
\end{aligned}$$

The syntax we have described is based on von der Beeck's syntax from [28], but with some important differences. The two main differences between ours and von der Beeck's: 1) we do not explicitly include a pointer to the currently active state as part of the term, and 2) inter-level transitions are specified by fully qualified state names rather than configuration sets. In von der Beeck's syntax, an inter-level transition is specified as $(\underline{t}, i, sr, e, a, td, j, ht)$ where $\underline{t}$ is the transition's name, i is the index of the source state, sr is the "source restriction", a (possibly incomplete) configuration, this is, a set of state names which determines the actual state inside the source which is the origin of the transition, e is the trigger, a is the action, td is the "target determinant", the set of state names which determines the actual state within the target state, whose index is j, and ht is the history type of the transition. This means that the actual origin of the transition is determined by the pair (i, sr) and the actual destination is given by the pair (j, td). In our syntax we represent the origin and destination of a transition by a sequence of enclosing state names, which can be thought of as a fully qualified name for a state.

2.3 kiltera's core: the $\kappa\lambda\tau$-calculus

In this Section we describe the core of the kiltera language, a subset called the $\kappa\lambda\tau$-calculus. This calculus is similar to the well-known π-calculus [15, 14], but departs from it in some important ways. In particular, the $\kappa\lambda\tau$-calculus allows the description of *timed* behaviour (cf. π-calculus processes are untimed). Communication is by asynchronous message passing, both by unicasting and multicasting (cf. communication in the π-calculus is only by unicasting). Unlike the π-calculus, there are two unicasting send operations: $x \uparrow v$ and $x!v$ (and two corresponding multicasting versions). The later ($x!v$) corresponds more closely to the π-calculus send action. The difference is in their behaviour with respect to the presence of listeners *at the time* the action is performed. In addition to these features, it also includes some higher-level capabilities such as primitive constants, data structures and pattern matching. These features facilitate the description of statechart behaviour.

The basic syntax, which defines the set KLT of $\kappa\lambda\tau$ terms, is as follows:

$$\begin{array}{rcl} P & ::= & \surd \mid \alpha \mid \Delta E \to P \mid \nu x.P \mid P_1 \parallel P_2 \\ & & \mid P_1; P_2 \mid \sum_{i \in I} \beta_i \to P_i \mid A(x_1, ..., x_n) \\ & & \mid \mathsf{match}\ E : F_1 \to P_1 | \cdots | F_n \to P_n \\ \alpha & ::= & x \uparrow E \mid x!E \mid x \uparrow^* E \mid x!^* E \qquad \beta \ ::= \ x?F\delta t \end{array}$$

where P, P_i range over *process terms*, α ranges over *actions*, β ranges over *input guards*, x, x_i range over the set of *(event or channel) names*, t ranges over the set of *(variable) names*, A ranges over the set of *process names*, E ranges over *expressions*, and F ranges over *patterns*. Process definitions have the form:

$$A(x_1, ..., x_n) \overset{def}{=} P$$

The syntax for expressions and patterns is as follows:

$$\begin{array}{rcl} E & ::= & F \mid op\, E \mid E_1\, op\, E_2 \mid f(E_1, ..., E_m) \\ F & ::= & n \mid \mathsf{true} \mid \mathsf{false} \mid \text{“}s\text{”} \mid x \mid (E_1, ..., E_m) \end{array}$$

where $op \in \{+, -, *, /, \mathsf{mod}, \mathsf{and}, \mathsf{or}, \mathsf{not}, <, >, =, <=, >=, !=\}$, n ranges over floating point numbers, s ranges over strings, x ranges over variable names, and f ranges over function names, with function definitions having the form:

$$f(x_1, ..., x_n) \overset{def}{=} E$$

The process $\surd$ simply terminates, and cannot interact with others. Processes α are output processes. The process $x \uparrow E$ triggers an event x and associates this event with the value of expression E. Alternatively, one can say that it sends the message E through channel x. This is a *transient trigger*, this is, if there are no *listeners*, i.e. processes ready to interact via x, at the current time, then the event is discarded. In any case, the trigger is "consumed" in the current time. This contrasts with $x!E$, which also triggers x with E as value, but if there are no listeners at the current time, this process remains alive until there is at least one other process ready to interact with it. Once interaction occurs, the trigger is consumed. We call $x!E$ a *lasting trigger*. Both $x \uparrow E$ and $x!E$

perform communication by unicasting. The processes $x \uparrow^* E$ and $x!^*E$ are the multicasting variants of $x \uparrow E$ and $x!E$ respectively, so the message is delivered to all relevant listeners. If there are no listeners when $x!^*E$ is performed, it will remain alive until at least one process is ready to accept the message. At that time, all potential receivers will obtain the message, and the trigger is consumed (not replicated in the future). In all of the trigger processes, the expression E is optional. The process $\Delta E \to P$ delays the execution of process P by an amount of time t, the value of the expression E. The process $\nu x.P$ hides the name x from the environment, so that it is private to P. Alternatively, $\nu x.P$ can be seen as the creation of a new name, i.e. a new event or channel, whose scope is P. We write $\nu x_1, x_2, ..., x_n.P$ for the process term $\nu x_1.\nu x_2....\nu x_n.P$. The process $P_1 \parallel P_2$ is the parallel composition of P_1 and P_2. In its generalized form, we write $\Pi_{i\in\{1,\ldots,n\}} P_i$ for $P_1 \parallel P_2 \parallel \cdots \parallel P_n$. The process $P_1; P_2$ is the sequential composition of P_1 and P_2, this is, P_1 must terminate before beginning P_2.[3] The process $\sum_{i\in I} \beta_i \to P_i$ is a *receiver* or *listener*, consisting of a list of alternative input guarded processes $\beta_i \to P_i$. Each *input guard* β_i is of the form $x_i?F_i\delta t_i$, where x_i is an event/channel name, F_i is a pattern, and t_i is a variable[4]. This process listens to all events (channels) 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 and the alternatives are discarded. A listener process represents, thus, a process in a state with external choice. Before P_i is executed, t_i is assigned the *elapsed time*, this is, the time the process remained blocked waiting for an event to occur. Furthermore, if the event x_i was triggered with some value v which matches the pattern F_i, this matching results in the binding of F_i's variables by the corresponding values of v.[5] The scope of these bindings is P_i. The suffixes F_i and δt_i are optional. If F_i is absent, no pattern-matching is done. Sometimes we write listeners in infix notation: $x_1?F_1\delta t_1 \to P_1 + \cdots + x_n?F_n\delta t_n \to P_n$. This operator is not commutative: a guard $x_i?F_i\delta t_i$ will be enabled only if the previous $i-1$ guards are not enabled (no events triggered or matched patterns). Hence the guards are evaluated in order. We will use the symbol _ in patterns as a nameless variable (i.e. a pattern that matches anything) to denote "anything else". The process $A(y_1, ..., y_n)$ creates a new instance of a process defined by $A(x_1, ..., x_n) \overset{def}{=} P$, where the ports $x_1, ..., x_n$ are substituted in the body P by the events or channels $y_1, ..., y_n$. Finally, the process $\mathsf{match}\ E : F_1 \to P_1 | \cdots | F_n \to P_n$ evaluates the expression E and attempts to match it with each pattern F_i. If a pattern F_i matches then the corresponding process P_i is executed. If more than one pattern matches the choice is non-deterministic. This construct is syntactic sugar for $\nu x.(x \uparrow E \parallel x?F_1 \to P_1 + \cdots + x?F_n \to P_n)$. We also write $\mathsf{match}\ E : |_{i\in I} F_i \to P_i$ for $\mathsf{match}\ E : F_1 \to P_1 | \cdots | F_n \to P_n$.

For a formal semantics of this calculus, we refer the reader to [22].

[3]The sequential composition operator is actually a derived operator, but for simplicity we include it here as a primitive. See [22] for details.

[4]The symbol δ is just a separator, which is read as "after".

[5]This is essentially the same as pattern-matching in functional languages like ML or Haskell.

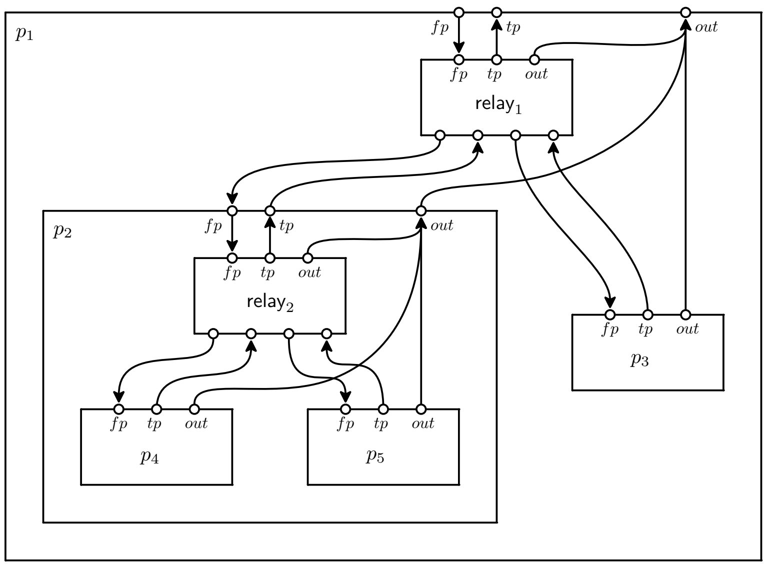

Figure 2: Hierarchical structure of the kiltera process corresponding to the statechart from Figure 1. Each box labelled p_i represents the kiltera process corresponding to the state named n_i.

3 From statecharts to kiltera processes

In this Section we will define a mapping $[\![\cdot]\!]_{T,fp,tp,out} : \mathsf{SC} \rightarrow \mathsf{KLT}$ which, for any well-formed statechart term gives a kiltera process describing the statechart's behaviour. When translating a term $s \in \mathsf{SC}$, the additional parameters specify the set of transitions (T) going out of s or any of its descendants and the ports (fp, tp, and out) used by the resulting kiltera term to communicate with the enclosing process (see Subsection 3.1 below for details on the role of these ports).

3.1 Structure of a mapped statechart

Each statechart is mapped to a kiltera process which mimics its nesting structure[6]. In the case of or-states and and-states, the corresponding kiltera process consists of one sub-process for each sub-state, and a *relay* process, which is in charge of controlling which sub-state is active, as well as forwarding messages accordingly. For example, the statechart in Figure 1 is mapped to a kiltera process with a structure depicted in Figure 2.

Each kiltera process denoting a statechart has an interface consisting of three ports: fp, tp and out. This is depicted in Figure 3. The first port, fp is used to receive messages from the parent, this is, the kiltera process representing the enclosing statechart. The second port, tp is used to send messages to the

[6]This is, the mapping is compositional.

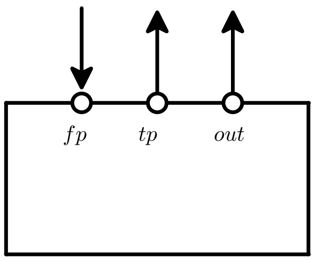

Figure 3: kiltera process interface for a statechart.

parent. The third port, *out* is used to send output messages (i.e. the action of the transition performed).

At run-time, the relay component of an or-state keeps track of the *currently active sub-state* by a pair of links fc and tc to the corresponding sub-process. Since the active sub-state changes, so do these links. Hence, channel mobility plays a fundamental role in describing the dynamics of statecharts.

The general idea is as follows: when a new event is received by the statechart, an *event* message is given to the top-level (a.k.a. the root) process for the statechart. Its relay then forwards this event message to its currently active sub-state, which in turn forwards it down its currently active sub-state, and so on. When the message reaches a basic state, it is matched against the triggers of the outgoing transitions from this state. If some transition matches the trigger, it means that the transition is taken, so an *entry* message is sent to the process representing the target of the transition, and the process representing the currently active state exits (executing its exit action) and becomes inactive. Since there is no direct link between the processes representing the source and target of the transition, this entry message is routed through the hierarchy of processes, according to the states that the transition crosses. While the entry message is sent to the destination, the states being exited execute their exit action and become inactive as the entry message travels "up" to the least common ancestor of the source and target of the transition. When the message reaches the least common ancestor, the transition's action is executed, and the message is routed "down" to the appropriate target. As the message goes "down", the states being entered become active and their entry actions executed, until we reach the actual target, which answers with an *entry acknowledgement* message, to be forwarded all the way up to the root. On the other hand, if the event did not match any transition of the currently active basic state, then a message is sent to the enclosing state (a.k.a. the parent) informing it that the event was not handled. In such case, the enclosing state attempts to match the event in the same manner as the basic state.

Message	Description
“enter”	enter a state
“exit”	exit from current state
$(\text{“}\mathsf{enter}\text{”}, doact, actdone)$	enter a state with action trigger (see sec. 3.4.4)
“$\mathsf{en\_ack}$”	enter acknowledgment
“$\mathsf{ex\_ack}$”	exit acknowledgment
$(\text{“}\mathsf{evt}\text{”}, value)$	event with given value
“enhh”	event not handled here

Table 1: Message types.

3.2 Messages and paths

Messages exchanged between processes representing states are of the form $(\vec{p}, m)$ where $\vec{p} = \langle p_1, ..., p_k \rangle$ is the *path* to the destination, and m is the message.

A message m contains either a command to enter or exit a state, an event, or an acknowledgment. The possible values are shown in Table 1.

A path $\vec{p}$ is a sequence of the form $\langle \text{“}\mathsf{up}\text{”}, \text{“}\mathsf{up}\text{”}, ..., \text{“}\mathsf{up}\text{”}, n_1, n_2, ..., n_l \rangle$ which describes how to route the message from one state to another. The prefix $\langle \text{“}\mathsf{up}\text{”}, ..., \text{“}\mathsf{up}\text{”} \rangle$ specifies the number of levels the message has to go “up” in the nesting hierarchy, and the postfix $\langle n_1, ..., n_l \rangle$ specifies the states that must be taken going “down” in the nesting hierarchy until reaching the destination. Here n_l is the name of the target. [7]

Example 2. Consider the statechart in Figure 4. The nesting tree for this statechart, and the path from state n_9 to state n_{10} are shown in Figure 5. If the statechart is currently in state n_9 and receives and event that causes transition $\underline{t}_1$ to n_{10} to fire, an event message (evt) is sent from the top-level state n_1 downwards to n_9 with path $\langle n_2, n_4, n_7, n_9 \rangle$. This causes state n_9 to send an entry message (“enter”) to n_{10} with path $\langle \text{“}\mathsf{up}\text{”}, \text{“}\mathsf{up}\text{”}, n_5, n_8, n_{10} \rangle$. It has only two “$\mathsf{up}$”’s because one is taken at state n_7 and the other is taken at state n_4, leaving the message in state n_2, where it will go down the path $\langle n_5, n_8, n_{10} \rangle$.

We define the following function to compute the path of a given transition from one state to another.

Definition 4. (Path of a transition) Let $t = (\underline{t}, \vec{o}, e, a, \vec{d}) \in \mathsf{TR}$ be some transition. The path from $\mathsf{src}(t)$ to $\mathsf{trg}(t)$ is given by:

$$\mathsf{tpath}(t) \overset{def}{=} \langle \text{“}\mathsf{up}\text{”} \rangle_{i \in \{1, ..., |\vec{o}|-1\}} \cdot \vec{d}$$

This is, the path of a transition contains an “up” for each state until the least common ancestor of the source and target, followed by the sequence of names going “down” until the destination.

[7] A message sent to a descendant would not have any “up”’s, but in our translation, messages sent to descendants are sent only to direct sub-states through their specific channels, and the path will be $\langle \rangle$.

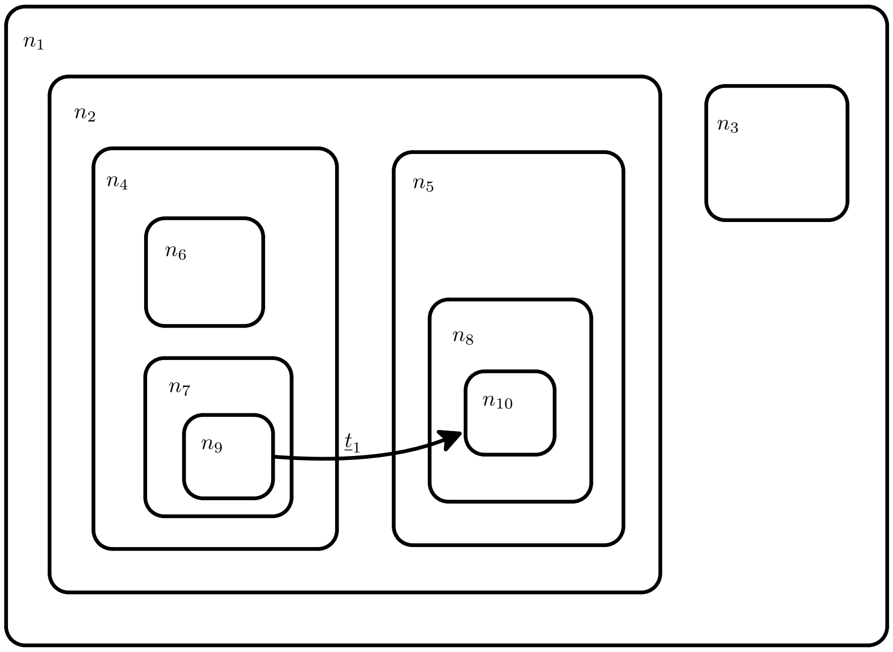

Figure 4: A statechart with an inter-level transition.

$$\begin{tikzcd}
 & & & n_1 \arrow[dl, no head] \arrow[dr, no head] & \\
 & & n_2 \arrow[dr] & & n_3 \\
 & n_4 \arrow[ur] \arrow[dl, no head] & & n_5 \arrow[d] & \\
n_6 & & n_7 \arrow[ul] & n_8 \arrow[d] & \\
 & & n_9 \arrow[u] & n_{10} &
\end{tikzcd}$$

Figure 5: Nesting tree and path from n_9 to n_{10}.

We also define the following functions to compute the path of a transition from one state to another, given the source and target of the transition.

3.3 Executing actions

When a state s is entered, its entry action en must be executed, and when it is exited, its exit action ex must be executed. Furthermore, when a transition is taken, its action must also be executed. In the remainder of this Section we assume there is a function exec which executes actions (entry, exit and transition actions). More precisely, it is a function $\mathsf{exec}_{out} : \mathcal{A} \to \mathsf{KLT}$ mapping actions to appropriate kiltera processes, which may produce its output at a given port out.[8]

Once a state has executed its entry action (resp. exit action), it sends an acknowledgment signal "en_ack" (resp. "ex_ack") to its parent. These acknowledgments are used to guarantee that actions are fully executed and in the right order, and to guarantee run-to-completion semantics.

3.4 Mapping basic states

In the remainder of this Subsection we will assume that $s = [n, (en, ex)]$ is the basic state that we are translating. Before showing the translation of s itself, we need some preliminary process definitions.

3.4.1 Entry sequence

The following process executes the state's entry action en and sends an entry acknowledgment to its parent.[9]

$$\mathsf{basic\_enseq}_s(tp, out) \overset{def}{=} \mathsf{exec}_{out}(en); tp \uparrow (\langle\rangle, \text{“}\mathsf{en\_ack}\text{”})$$

3.4.2 Exit sequence

The following process is analogous. It executes the state's exit action ex and sends an exit acknowledgment to its parent.

$$\mathsf{basic\_exseq}_s(tp, out) \overset{def}{=} \mathsf{exec}_{out}(ex); tp \uparrow (\langle\rangle, \text{“}\mathsf{ex\_ack}\text{”})$$

[8]In this encoding we assume that the only observable effect of executing an action is sending a message to an object, and thus, we only assume the (possible) use of an out port. A more comprehensive mapping would also allow actions to modify the attributes of the object who owns the statechart. In such case, the $exec$ mapping would also be parametrized by some link(s) to a process that gives access to the object's attributes.

[9]Note that the path is the empty sequence $\langle\rangle$ since the destination is the parent itself.

3.4.3 Inactive state

The following process represents an inactive basic state. The subscript T denotes the set of outgoing transitions from this state (s).

$$\begin{aligned}
&\mathsf{basic\_inactive}_{s,T}(fp, tp, out) \overset{def}{=} \\
&\qquad fp?(\langle\rangle, \text{“}\mathsf{enter}\text{”}) \rightarrow (\mathsf{basic\_enseq}_s(tp, out); \mathsf{basic\_active}_{s,T}(fp, tp, out))
\end{aligned}$$

An inactive basic state can only accept enter messages. Once an enter message arrives, it executes its entry sequence and becomes active.

3.4.4 Active state

The following process represents an active basic state. As before, the subscript T denotes the set of outgoing transitions from this state.

$$\begin{aligned}
&\mathsf{basic\_active}_{s,T}(fp, tp, out) \overset{def}{=} \\
&\quad fp?(\langle\rangle, \text{“}\mathsf{exit}\text{”}) \rightarrow \\
&\qquad (\mathsf{basic\_exseq}_s(tp, out); \\
&\qquad\ \ \mathsf{basic\_inactive}_{s,T}(fp, tp, out)) \\
&\quad + \textstyle\sum_{t_i \in T} fp?(\langle\rangle, (\text{“}\mathsf{evt}\text{”}, \mathsf{evt}(t_i))) \rightarrow \\
&\qquad (\mathsf{basic\_exseq}_s(tp, out); \\
&\qquad\ \ \mathsf{basic\_jump}_{s,T,t_i}(fp, tp, out)) \\
&\quad + fp?(\langle\rangle, (\text{“}\mathsf{evt}\text{”}, \_)) \rightarrow \\
&\qquad (tp\!\uparrow\!(\langle\rangle, \text{“}\mathsf{enhh}\text{”}); \\
&\qquad\ \ \mathsf{basic\_active}_{s,T}(fp, tp, out))
\end{aligned}$$

An active state can receive either an exit message (“exit”) or an event of the form (“evt”, $value$). If it receives an exit message, it must perform the exit sequence and become inactive. If it receives an event, there are two possibilities: either the event's $value$ matches the trigger $\mathsf{evt}(t_i)$ of some transition t_i, or it doesn't.

If the event's $value$ matches the trigger $\mathsf{evt}(t_i)$ of some transition t_i, the state must execute the exit sequence, and then handle the event, which is done according to the following:

$$\begin{aligned}
&\mathsf{basic\_jump}_{s,T,t_i}(fp, tp, out) \overset{def}{=} \\
&\quad \nu\, doact, actdone.\, (doact? \rightarrow (\mathsf{exec}_{out}(\mathsf{act}(t_i));\ actdone\!\uparrow) \\
&\qquad\qquad\qquad\qquad \parallel tp\!\uparrow\!(\mathsf{tpath}(t_i), (\text{“}\mathsf{enter}\text{”}, doact, actdone))); \\
&\quad \mathsf{basic\_inactive}_{s,T}(fp, tp, out)
\end{aligned}$$

This process sets up the execution of the transition's action ($\mathsf{act}(t_i)$) but this is not executed right away, since there might be exit actions of enclosing states that must be executed before. Instead, the execution of this action is deferred until all necessary exit actions of enclosing states are executed. Hence we only create a “callback” process $doact? \rightarrow (\mathsf{exec}_{out}(\mathsf{act}(t_i)); actdone\!\uparrow)$ that waits for the signal *doact* which is to be triggered when the action must be executed, and then triggers signal *actdone* when the action has finished. While creating this "callback" for the action, we send an "enter" message to the destination, via the state's parent (tp), following the appropriate path $\mathsf{tpath}(t_i)$. This message carries additional information, namely the signals *doact* and *actdone*, so that the action is triggered at the appropriate moment, when all "up" steps have been performed, and just before entering states going down.[10] Finally the state becomes inactive.

If the state receives an event which does not match the trigger event of any of the outgoing transitions, then it sends the parent an "enhh" message indicating that the event is not handled here. Then it remains active.

3.4.5 Translation of a basic state

Given these definitions we are now able to map a basic state $s = [n, (ex, en)]$ to a kiltera term as follows:

$$[\![[n, (ex, en)]]\!]_{T,fp,tp,out} \overset{def}{=} \mathsf{basic\_inactive}_{[n,(ex,en)],T}(fp, tp, out)$$

This is, a state is mapped to an initially inactive process.

3.5 Mapping or-states

In the remainder of this Subsection we will assume that $s = [n, s_{1..k}, T, (en, ex)]$ is the or-state that we are translating.

3.5.1 Entry sequence

The entry sequence for an or-state must first execute the state's entry action en, then recursively enter the default state (with the process enter_child defined below), and finally send an acknowledgment to the parent:

$$\begin{aligned}&\mathsf{or\_enseq}_s(tp, out, fd, td) \overset{def}{=} \\ &\qquad \mathsf{exec}_{out}(en); \mathsf{enter\_child}(fd, td); tp\!\uparrow(\langle\rangle, \text{“}\mathsf{en\_ack}\text{”})\end{aligned}$$

where fd and td are the ports from the default state and to the default state respectively, tp is the port to the parent, and out is the output port.

The process enter_child tells a given child to enter, and then waits for acknowledgment:

$$\mathsf{enter\_child}(fc, tc) \overset{def}{=} tc\!\uparrow(\langle\rangle, \text{“}\mathsf{enter}\text{”});\ fc?(\langle\rangle, \text{“}\mathsf{en\_ack}\text{”})$$

Here fc denotes the port where messages from the child are expected, and tc the port where messages to the child are sent.

[10]Note that here we make use of kiltera's (and the π-calculus) defining characteristic: channel mobility. We send an event (channel) as part of the message.

Note that the default child must be fully entered and given its entry acknowledgment, before the or-state sends its own acknowledgment to its parent. This guarantees that when an entry acknowledgment is received, the corresponding child has completed its entry and executed all entry actions in the appropriate order.

3.5.2 Exit sequence

The exit sequence of an or-state is analogous to its entry sequence, but when the state is asked to exit, its *currently active sub-state* must be exited (using the process exit_child defined below) before the state's exit action is performed.

$$\mathsf{or\_exseq}_s(tp, out, fc, tc) \overset{def}{=} \\ \mathsf{exit\_child}(fc, tc); \mathsf{exec}_{out}(ex); tp\uparrow(\langle\rangle, \text{“}\mathsf{ex\_ack}\text{”})$$

The exit_child process is analogous to the enter_child process:

$$\mathsf{exit\_child}(fc, tc) \overset{def}{=} tc\uparrow(\langle\rangle, \text{“}\mathsf{exit}\text{”});\ fc?(\langle\rangle, \text{“}\mathsf{ex\_ack}\text{”})$$

where fc and tc are the channels to the currently active child.

As before, the currently active sub-state must have fully exited before the exit action of the state is performed and the exit acknowledgment sent to the parent. This guarantees that when an exit acknowledgment is received, the corresponding child has completed its exit and executed all exit actions in the appropriate order.

3.5.3 Forward sequence

When we are taking a transition whose target is a descendant of the state, rather than the state itself, the relay for the state must forward the entry message to the appropriate child, in accordance with the message's path. This is accomplished by the following process, which sends a message to the child specified by the ports fc and tc with etc as the *remaining* path to the destination[11], and waits for an entry acknowledgment from the child which, in turn, is forwarded up to the state's parent.

$$\mathsf{forward\_enter}(tp, fc, tc, etc) \overset{def}{=} \\ tc\uparrow(etc, \text{“}\mathsf{enter}\text{”});\ fc?(\langle\rangle, \text{“}\mathsf{en\_ack}\text{”}) \rightarrow tp\uparrow(\langle\rangle, \text{“}\mathsf{en\_ack}\text{”})$$

3.5.4 The relay

The relay process is in charge of keeping track of active sub-states, as well as handling and forwarding messages. The relay can be in two modes: active or

[11]The first element of the path is stripped by the relay, as described below.

inactive. Initially it starts in the inactive state:

$$\mathsf{or\_relay}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k}) \stackrel{def}{=} \mathsf{or\_inactive}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k})$$

The relay contains not only the fp, tp, and out ports of the state, but also a pair of ports fc_i and tc_i for each sub-state i to communicate with them[12]. We assume that the first component represents the default state, and therefore fc_1 and tc_1 are the links to the default state. The subscript T denotes the set of outgoing transitions from this state (s) or any descendant.

3.5.5 Inactive state

When the state is inactive, it may receive an enter message directed to it (if the state is the target of the corresponding transition), or to a descendant (if the descendant is the target of the transition). In the following, fp, tp, and out are the usual ports. In addition to these, for each child i we have a pair of ports fc_i and tc_i respectively from the child and to the child.

$$\begin{aligned}
&\mathsf{or\_inactive}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k}) \stackrel{def}{=} \\
&\quad fp?(\langle\rangle, \text{“enter”}) \rightarrow \\
&\qquad (\mathsf{or\_enseq}_s(tp, out, fc_1, tc_1); \\
&\qquad\ \mathsf{or\_active}_{s,T}(fp, tp, out, fc_1, tc_1, fc_{1..k}, tc_{1..k})) \\
&\quad + \textstyle\sum_{s_i \in s_{1..k}} fp?(\langle \mathsf{name}(s_i)\rangle \cdot etc, \text{“enter”}) \rightarrow \\
&\qquad (\mathsf{exec}_{out}(en); \\
&\qquad\ \mathsf{forward\_enter}(tp, fc_i, tc_i, etc); \\
&\qquad\ \mathsf{or\_active}_{s,T}(fp, tp, out, fc_i, tc_i, fc_{1..k}, tc_{1..k}))
\end{aligned}$$

When the inactive state receives an enter message directed to it, is executes its entry sequence (using the ports fc_1 and tc_1 to communicate with the default state), and becomes active, setting the currently actuve sub-state to be the default state by executing the $\mathsf{or\_active}_{s,T}$ process with the ports fc_1 and tc_1 as the fourth and fifth ports respectively.

When the inactive state receives a message aimed at a descendant within the sub-state s_i, it must be an enter message to the descendant, and thus, the state must be entered. This means that the enter action must be executed, the message must be forwarded to the corresponding sub-process (via the channels fc_i and tc_i which correspond to the child s_i),[13] and then it must become active, with the sub-state s_i active (indicated by the fourth and fifth ports of the $\mathsf{or\_active}_{s,T}$ process). By executing the entry action before forwarding the message we guarantee the semantics of transitions, where entry actions must be executed in order of nesting, from outermost to innermost, for all states that the transition is entering.

[12] Note that we use the sequence notation $fc_{1..k}$ for the sequence of ports $fc_1, ..., fc_k$, and similarly for $tc_{1..k}$.

[13] Note that at this point the first item of the path is stripped and only the remainder etc is passed to the forwarding process.

3.5.6 Active state

The following process represents an active or-state. As before, for each child i we have a pair of ports fc_i and tc_i respectively from the child and to the child, and the distinguished ports fc and tc link the process with the currently active sub-state.[14]

$$
\begin{aligned}
&\mathsf{or\_active}_{s,T}(fp, tp, out, fc, tc, fc_{1..k}, tc_{1..k}) \overset{def}{=} \\
&\quad fp?(\langle\rangle, \text{“}\mathsf{exit}\text{”}) \rightarrow \\
&\qquad (\mathsf{or\_exseq}_s(tp, out, fc, tc); \\
&\qquad\ \mathsf{or\_inactive}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k})) \\
&\quad + fp?(\langle\rangle, (\text{“}\mathsf{evt}\text{”}, x)) \rightarrow \\
&\qquad \mathsf{or\_handle\_event}_{s,T}(fp, tp, out, fc, tc, fc_{1..k}, tc_{1..k}, x)
\end{aligned}
$$

When an or-state is active it may receive an exit message or an event. If it is an exit message, it must execute the exit sequence, sending an exit message to the currently active sub-state (via tc) and then become inactive. If it is an event, the $\mathsf{or\_handle\_event}_{s,T}$ process, defined below, takes care of it.

3.5.7 Handling events

The semantics of UML statecharts states that when two transitions with the same trigger are enabled, and the source of one is a descendant of the other (i.e. it is at a lower level of nesting), then the one with lower level takes priority over its ancestor. In order to capture this semantics, the event handling process must first send the event message "down" the nesting hierarchy to the currently active sub-state to give it a chance to handle it. Suppose the or-state s receives an event. Then, the event is sent to the currently active sub-state. There are two possibilities, either:

1. The event is handled by the currently active sub-state, i.e., there was a transition with a trigger that matched the event, or
2. The event is not handled by the currently active sub-state.

In the first possibility, we have two cases, depending on the target of the transition: either the target of the transition is *outside* s (Figure 6 (a)), or the target is inside s. Furthermore, in the later case we have two sub-cases: either the transition's source and target are in different sub-states of s (Figure 6 (b)), or they are in the same sub-state (Figure 6 (c)).

For each of these possible scenarios we have a corresponding answer from the currently active child:

1. If the event was handled by the child (or some descendant), with some transition from the child (or a descendant) to some state *outside* s, then the child responded with an enter message directed to the target state *outside* s, with a path that begins with "up",

[14]Note that at run-time these ports will link the relay with different sub-states, as the active sub-state changes. This exemplifies our use of channel mobility.

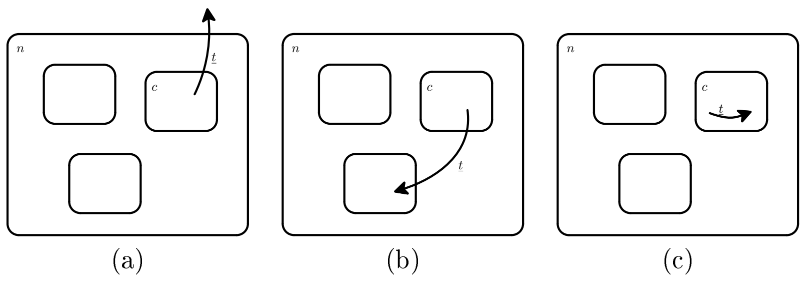

Figure 6: Event handled by the currently active sub-state c.

2. If the event was handled by the child (or some descendant), with some transition from the child (or a descendant) to some state *inside* s, where the source and target of the transition are in *different* sub-states of s, and so s is the least common ancestor of the source and target of the transition, then the child responded with an enter message directed at the target state *inside* s, with a path that begins with the name of the (ancestor of the) target state,

3. If the event was handled by the child (or some descendant), with some transition from the child (or a descendant) to some state *inside* s, where the source and target of the transition are in the *same* sub-state of s, and so s is not the least common ancestor of the source and target of the transition, then, the child responded with an entry acknowledgment ("en_ack"), sent by the target of the transition,

4. If the event was not handled by the child (or any descendant), then, the child responded with an "enhh" message.

Each of these alternatives is handled by the branches of the choice below.

$$
\begin{array}{l}
\mathsf{or\_handle\_event}_{s,T}(fp, tp, out, fc, tc, fc_{1..k}, tc_{1..k}, x) \stackrel{def}{=} \\
\quad tc\!\uparrow\!(\langle\rangle, (\text{“}\mathsf{evt}\text{”}, x)); \\
\quad\ (fc?(\langle\text{“}\mathsf{up}\text{”}\rangle \cdot etc, msg) \rightarrow \\
\qquad\quad \mathsf{or\_trans\_out}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k}, etc, msg) \\
\quad\ + \sum_{s_i \in s_{1..k}} fc?(\langle \mathsf{name}(s_i)\rangle \cdot etc, (\text{“}\mathsf{enter}\text{”}, doact, actdone)) \rightarrow \\
\qquad\quad \mathsf{or\_trans\_sibling}_{s,T,i}(fp, tp, out, fc_{1..k}, tc_{1..k}, doact, actdone, etc) \\
\quad\ + fc?(\langle\rangle, \text{“}\mathsf{en\_ack}\text{”}) \rightarrow \\
\qquad\quad \mathsf{or\_trans\_internal}_{s,T}(fp, tp, out, fc, tc, fc_{1..k}, tc_{1..k}) \\
\quad\ + fc?(\langle\rangle, \text{“}\mathsf{enhh}\text{”}) \rightarrow \\
\qquad\quad \mathsf{or\_match\_event}_{s,T'}(fp, tp, out, fc, tc, fc_{1..k}, tc_{1..k}, x))
\end{array}
$$

1. In the first case, the child answers with an enter message directed to a state outside s, and therefore the path begins with "up". This case is handled by the process $\mathsf{or\_trans\_out}_{s,T}$ defined below. We must exit s which entails executing its exit action ex, then send the message to the

parent, stripping the first item from the path (so the remaining path is etc), and then become inactive.

$$\begin{aligned}&\mathsf{or\_trans\_out}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k}, etc, msg) \overset{def}{=}\\ &\quad \mathsf{exec}_{out}(ex);\\ &\quad tp\!\uparrow\!(etc, msg);\\ &\quad \mathsf{or\_inactive}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k})\end{aligned}$$

2. In the second case, the child answers with an enter message directed to a state inside s (different than the source of the transition), and therefore the path begins with the name of a sub-state of s, specifically the name of the sub-state s_i which contains the target of the message. This case is handled by the process $\mathsf{or\_trans\_sibling}_{s,T,i}$ defined below.

   $$\begin{aligned}&\mathsf{or\_trans\_sibling}_{s,T,i}(fp, tp, out, fc_{1..k}, tc_{1..k}, doact, actdone, etc) \overset{def}{=}\\ &\quad doact\!\uparrow;\\ &\quad actdone? \rightarrow (\mathsf{forward\_enter}(tp, fc_i, tc_i, etc);\\ &\qquad\qquad\qquad \mathsf{or\_active}_{s,T}(fp, tp, out, fc_i, tc_i, fc_{1..k}, tc_{1..k}))\end{aligned}$$

   The state s must be the least common ancestor of the source and the target of the transition taking place. Hence the message is of the form (“enter”, $doact, actdone$) where $doact$ is the signal to execute the transition's action, and $actdone$ is the event signaling the termination of the action. At this point all relevant exit actions for the transition have been executed, so we trigger the action signal $doact$ and wait for the transition's action to finish. Then we forward the enter message to the appropriate target tc_i. The state remains active in this case, but the links to the currently active sub-state change to fc_i and tc_i.

3. In the third case, the child answers with an entry acknowledgment which indicates that the transition was fully handled within some sub-state of s. This case is handled by the process $\mathsf{or\_trans\_internal}_{s,T}$ below.

   $$\begin{aligned}&\mathsf{or\_trans\_internal}_{s,T}(fp, tp, out, fc, tc, fc_{1..k}, tc_{1..k}) \overset{def}{=}\\ &\quad tp\!\uparrow\!(\langle\rangle, \text{“en\_ack”});\\ &\quad \mathsf{or\_active}_{s,T}(fp, tp, out, fc, tc, fc_{1..k}, tc_{1..k})\end{aligned}$$

   In this case, we simply propagate the acknowledgment up to s's parent, and remain active, with the currently active sub-state unchanged.

4. In the last case, the child answers with an “enhh” message, indicating that the current sub-state did not handle the event. In this case, we must try to match the event, according to the transitions T' going out of s:

   $$T' \overset{def}{=} \{t \in T \mid \mathsf{src}(t) = \mathsf{name}(s)\}$$

   The matching of the event is done by the $\mathsf{or\_match\_event}_{s,T'}$ process:

$$\begin{array}{l}
\mathsf{or\_match\_event}_{s,T}(fp, tp, out, fc, tc, fc_1, ..., fc_k, tc_1, ..., tc_k, x) \overset{def}{=} \\
\quad \mathsf{match}\ x: \\
\quad |_{t_i \in T}\ \mathsf{evt}(t_i) \rightarrow \\
\qquad (\mathsf{or\_exseq}_s(tp, out, fc, tc); \\
\qquad\ \mathsf{or\_jump}_{s,T,t_i}(fp, tp, out, fc_{1..k}, tc_{1..k})) \\
\quad |\ \_ \rightarrow \\
\qquad (tp\!\uparrow(\langle\rangle, \text{“}\mathsf{enhh}\text{”}); \\
\qquad\ \mathsf{or\_active}_{s,T}(fp, tp, out, fc, tc, fc_1, ..., fc_k, tc_1, ..., tc_k))
\end{array}$$

When the event of transition t_i is matched, we must leave the state and therefore, we must execute the exit sequence (using the links fc and tc to communicate with the currently active sub-state), and then we jump out of the state, as specified by the following:

$$\begin{array}{l}
\mathsf{or\_jump}_{s,T,t_i}(fp, tp, out, fc_{1..k}, tc_{1..k}) \overset{def}{=} \\
\quad \nu\, doact, actdone.\, (doact? \rightarrow (\mathsf{exec}_{out}(\mathsf{act}(t_i));\ actdone\!\uparrow) \\
\qquad\qquad\qquad \parallel tp\!\uparrow(\mathsf{tpath}(t_i), (\text{“}\mathsf{enter}\text{”}, doact, actdone))); \\
\quad \mathsf{or\_inactive}_{s,T}(fp, tp, out, fc_1, tc_1, fc_{1..k}, tc_{1..k})
\end{array}$$

As in a basic state, we setup a “callback” process to execute the transition's action once a signal *doact* is triggered. Then we send the enter message to the target of the transition via the state's parent, and finally become inactive.

If none of the triggers match, we send an “enhh” message to the parent, while remaining active, to inform the parent that the event was not handled here.

3.5.8 Translation of an or-state

The translation of the or-state $s = [n, s_{1..k}, T, (en, ex)]$ consists of the relay and the parallel composition of the translations of each of the sub-state terms:

$$\begin{array}{l}
[\![s]\!]_{T,fp,tp,out} \overset{def}{=} \\
\quad \nu fc_1, ..., fc_k, tc_1, ..., tc_k.(\mathsf{or\_relay}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k) \\
\qquad\qquad\qquad \parallel \Pi_{i\in\{1,...,k\}}[\![s_i]\!]_{T_i,tc_i,fc_i,out})
\end{array}$$

where $T_i \overset{def}{=} \{t \in T \,|\, \mathsf{src}(t) \in \mathsf{descendants}(s_i) \cup \{\mathsf{name}(s_i)\}\}$ for each $i \in \{1, ..., k\}$ is the set of transitions with source in sub-state s_i, i.e. going out of s_i or any of its descendants. Note that the parameters of the translation $[\![s_i]\!]$ of a sub-state s_i are given in order T, tc_i, fc_i, out. This is because the relay's tc_i port must be linked to the child's fp port, and the fc_i port must be linked to the child's tp port.

3.6 Mapping and-states

And-states consists of a set of *orthogonal regions* each of which is a statechart in its own right. The kiltera process representing an and-state has the same structure described above, with a relay and a process corresponding to each orthogonal region. The main difference with or-states is that in an or-state there is only one active sub-state at any point in time whereas in an and-state all orthogonal components are active. When an event arrives, it is broadcasted to all orthogonal regions.

In this report we deal only with a subset of the full UML State Machines specification. In particular we do not deal with forks and joins. As a consequence, if the target of a transition is a sub-state of an and-state, it is only within one of its orthogonal regions, so that when the transition is taken, the other orthogonal regions will enter through their default state. Furthermore, when a transition comes out of a sub-state of an and-state, all other regions will exit.

3.6.1 Entry sequence

When entering an and-state, all orthogonal regions must be entered, but not before executing the state's own entry action.

$$\begin{aligned}&\mathsf{and\_enseq}_s(tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k) \overset{def}{=} \\ &\qquad \mathsf{exec}_{out}(en); (\Pi_{i\in 1..k}\mathsf{enter\_child}(fc_i, tc_i)); tp\!\uparrow\!(\langle\rangle, \text{``en\_ack''})\end{aligned}$$

where enter_child is the same as in Section 3.5.1. Note that the parallel composition of enter_child processes, which causes all orthogonal regions to enter, does not prescribe a particular order of entry to the orthogonal regions. This is conformant with the UML specification which leaves open the particular order of entry.

3.6.2 Exit sequence

The exit sequence for the and-state is analogous. All sub-regions are ordered to exit before executing the state's own exit action.

$$\begin{aligned}&\mathsf{and\_exseq}_s(tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k) \overset{def}{=} \\ &\qquad (\Pi_{i\in 1..k}\mathsf{exit\_child}(fc_i, tc_i)); \mathsf{exec}_{out}(ex); tp\!\uparrow\!(\langle\rangle, \text{``ex\_ack''})\end{aligned}$$

As before, exit_child is the same as in Section 3.5.2. Furthermore, the sequence operator guarantees that all enter_child sub-processes will be fully completed before the execution of the state's exit action.

3.6.3 Forward sequence

The forward sequence for and-states is the same as for or-states, as described in Section 3.5.3.

3.6.4 The relay

The relay for an and-state is analogous to the relay of an or-state. The relay can be in two modes: active or inactive. Initially it starts in the inactive state:

$$\begin{aligned}&\mathsf{and\_relay}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k) \overset{def}{=} \\ &\qquad\qquad \mathsf{and\_inactive}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k)\end{aligned}$$

The relay contains not only the fp, tp, and out ports of the state, but also a pair of ports fc_i and tc_i for each sub-state i to communicate with them. The subscript T denotes the set of outgoing transitions from this state (s) or any descendant.

3.6.5 Inactive state

The inactive-state of an and-state is similar to that of an or-state. It can receive either an enter message, in which case it performs the entry sequence and becomes active, or it can receive a message aimed at a sub-state s_i, in which case it must execute the entry action, forward the message to s_i, *and* enter the orthogonal regions to their default state. Once all orthogonal regions have entered, the state becomes active.

$$\begin{aligned}&\mathsf{and\_inactive}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k) \overset{def}{=} \\ &\quad fp?(\langle\rangle, \text{“}\mathsf{enter}\text{”}) \rightarrow \\ &\qquad (\mathsf{and\_enseq}_s(tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k); \\ &\qquad\ \mathsf{and\_active}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k)) \\ &\quad + \textstyle\sum_{s_i \in s_{1..k}} fp?(\langle \mathsf{name}(s_i)\rangle \cdot etc, \text{“}\mathsf{enter}\text{”}) \rightarrow \\ &\qquad (\mathsf{exec}_{out}(en); \\ &\qquad\ (\mathsf{forward\_enter}(tp, fc_i, tc_i, etc) \\ &\qquad\quad \parallel \Pi_{j \in 1..k \setminus \{i\}} \mathsf{enter\_child}(fc_j, tc_j)); \\ &\qquad\ \mathsf{and\_active}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k))\end{aligned}$$

Notice that the enter_child process is invoked for each child $j \in 1..k\setminus\{i\}$, this is, for every child which is not i, since this is the target of the transition, and all other orthogonal regions enter to their default state.

3.6.6 Active state

An active and-state is analogous to an or-state, but there is no distinguished currently active sub-state, since all orthogonal regions are active.

$$\begin{aligned}&\mathsf{and\_active}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k) \overset{def}{=} \\ &\quad fp?(\langle\rangle, \text{“}\mathsf{exit}\text{”}) \rightarrow \\ &\qquad (\mathsf{and\_exseq}_s(tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k); \\ &\qquad\ \mathsf{and\_inactive}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k)) \\ &\quad + fp?(\langle\rangle, (\text{“}\mathsf{evt}\text{”}, x)) \rightarrow \\ &\qquad\ \mathsf{and\_handle\_event}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k, x)\end{aligned}$$

As with an or-state, when an and-state is active it may receive an exit message or an event. If it is an exit message, it must execute the exit sequence, sending an exit message to all sub-states and then become inactive. If it is an event, the $\mathsf{and\_handle\_event}_{s,T}$ process, defined below, takes care of it.

3.6.7 Handling events

Handling events in an and-state is a bit different. The event must be broadcasted to all orthogonal regions, and there are four possible outcomes:

1. No region handled the event, and thus the and-state itself must attempt to handle it, or
2. Some regions handled the event, and there is at least one transition to be taken outside of s. [15]
3. Some (or all) regions handled the event *internally*, but no region that handled the event is performing a transition *outside* s.

To handle these possibilities, we use the following auxiliary processes:

$$\mathsf{broadcast}(x, tc_1, ..., tc_k) \overset{def}{=} (\Pi_{i \in 1..k} tc_i \uparrow (\langle\rangle, (\text{“}\mathsf{evt}\text{”}, x)))$$

This process sends the event x to all children (i.e. orthogonal regions).

$$\mathsf{wait\_for\_enack}(fc_1, ..., fc_k, all, n_1, ..., n_k) \overset{def}{=} \\ (\Pi_{i \in 1..k}(fc_i?(\langle\rangle, \text{“}\mathsf{enhh}\text{”}) \rightarrow n_i \uparrow + fc_i?(\langle\rangle, \text{“}\mathsf{en\_ack}\text{”}) \rightarrow \surd)); all \uparrow$$

This process waits for an entry acknowledgment or an “enhh” message from all children, and once all have answered, it triggers the *all* event. When the child i answers with “enhh”, an event n_i is triggered, in order to identify case 1 above, which is detected by the following process.

$$\mathsf{wait\_for\_enhh}(n_1, ..., n_k, nche) \overset{def}{=} (\Pi_{i \in 1..k} n_i? \rightarrow \surd); nche \uparrow$$

This process triggers the signal *nche* (no child handled the event) if (and only if) all events n_i where triggered.

With these processes, we can built the event handler for and-states:

[15] In the present version of this mapping we assume that at most one transition fired goes outside the state. Otherwise the and-state would act as a fork.

```
$\mathsf{and\_handle\_event}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k, x) \overset{def}{=}$
  $\mathsf{broadcast}(x, tc_1, ..., tc_k);$
   $\nu\, nche, all, n_1, ..., n_k.$
     $(\mathsf{wait\_for\_enack}(fc_1, ..., fc_k, all, n_1, ..., n_k)$
      $\parallel \mathsf{wait\_for\_enhh}(n_1, ..., n_k, nche)$
      $\parallel (nche? \rightarrow$
            $\mathsf{and\_match\_event}_{s,T'}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k, x)$
        $+ \sum_{i \in 1..k} fc_i?(\langle \text{“}\mathsf{up}\text{”} \rangle \cdot etc, msg) \rightarrow$
            $\mathsf{and\_trans\_out}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k}, etc, msg)$
        $+ all? \rightarrow$
            $(tp \uparrow (\langle\rangle, \text{“}\mathsf{en\_ack}\text{”});$
             $\mathsf{and\_active}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k))))$
```

First we broadcast the event to all children. Then we wait for their response, which can be an entry acknowledgment (“en_ack”), and event not-handled here message (“enhh”) or a message with a path beginning with “up”.

If all answered “enhh”, then we are in case 1, and so the signal *nche* is triggered, in which case we attempt to match the event with the transitions coming out of s, with the process $\mathsf{and\_match\_event}_{s,T'}$ were

$$T' \overset{def}{=} \{t \in T \mid \mathsf{src}(t) = \mathsf{name}(s)\}$$

is the set of transitions coming out of s. The matching of events is as for or-states:

```
$\mathsf{and\_match\_event}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k, x) \overset{def}{=}$
   $\mathsf{match}\ x:$
   $|_{t_i \in T}\ \mathsf{evt}(t_i) \rightarrow$
      $(\mathsf{and\_exseq}_s(tp, out, fc_1, ...., fc_k, tc_1, ..., tc_k);$
       $\mathsf{and\_jump}_{s,T,t_i}(fp, tp, out, fc_{1..k}, tc_{1..k}))$
   $|\ \_ \rightarrow$
      $(tp \uparrow (\langle\rangle, \text{“}\mathsf{enhh}\text{”});$
       $\mathsf{and\_active}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k))$
```

When the event of transition t_i is matched, we must execute the exit sequence, as we are leaving the state, and then it leaves the state according to the following:

```
$\mathsf{and\_jump}_{s,T,t_i}(fp, tp, out, fc_{1..k}, tc_{1..k}) \overset{def}{=}$
   $\nu\, doact, actdone.\ (doact? \rightarrow (\mathsf{exec}_{out}(\mathsf{act}(t_i));\ actdone \uparrow)$
                    $\parallel tp \uparrow (\mathsf{tpath}(t_i), (\text{“}\mathsf{enter}\text{”}, doact, actdone)));$
   $\mathsf{and\_inactive}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k})$
```

As before, we setup a “callback” process to execute the transition’s action once a signal *doact* is triggered. Then we send the enter message to the target of the transition via the state’s parent, and finally become inactive. If none of the triggers match, we send an “enhh” message to the parent, while remaining active, to inform the parent that the event was not handled here.

If someone answered with a message with a path beginning with "up", we are in case 2, taking a transition outside s. Hence we tell all children to exit, and after they have exited, we execute the state's exit action, send the message up to the parent and become inactive. This is performed by the following:

$$\begin{aligned}&\mathsf{and\_trans\_out}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k}, etc, msg) \overset{def}{=}\\ &\quad (\Pi_{j \in 1..k \setminus \{i\}} \mathsf{exit\_child}(fc_j, tc_j));\\ &\quad \mathsf{exec}_{out}(ex);\\ &\quad tp\!\uparrow\!(etc, msg);\\ &\quad \mathsf{and\_inactive}_{s,T}(fp, tp, out, fc_{1..k}, tc_{1..k},)\end{aligned}$$

Finally if all events answered, but the signal *nche* has not been triggered and no child answered with a message directed outside, then the signal *all* must have been triggered (case 3). In this case we simply send an entry acknowledgment to the parent, indicating the event was handled, and remain active.

3.6.8 Translation of an and-state

As with or-states, the translation of the and-state $s = [n, s_{1..k}, (en, ex)]$ consists of the relay and the parallel composition of the translations of each of the sub-state terms:

$$\begin{aligned}&[\![s]\!]_{T,fp,tp,out} \overset{def}{=}\\ &\quad \nu fc_1, ..., fc_k, tc_1, ..., tc_k.(\mathsf{and\_relay}_{s,T}(fp, tp, out, fc_1, ..., fc_k, tc_1, ..., tc_k)\\ &\qquad\qquad\qquad \parallel \Pi_{i \in \{1,...,k\}} [\![s_i]\!]_{T_i, tc_i, fc_i, out})\end{aligned}$$

where $T_i \overset{def}{=} \{t \in T \mid \mathsf{src}(t) \in \mathsf{descendants}(s_i) \cup \{\mathsf{name}(s_i)\}\}$ for each $i \in \{1, ..., k\}$ is the set of transitions with source in sub-state s_i, i.e. going out of s_i or any of its descendants.

4 Conclusions

We have presented a mapping from a significant subset of UML State Machines to a process algebra named kiltera. This constitutes a precise, compositional and executable formal semantics for UML State Machines.

While our mapping does not deal with several features such as the history mechanism or defered events, we preview that it will be relatively simple to modify it to support them. Furthermore, we have left out a precise description of object interaction, but this can be easily adapted: an object can be represented as a kiltera process with a suitable queueing mechanism for events, which dispatches them to the top-level state. The entry/exit acknowledgement protocol can be used to guarantee the run-to-completion semantics by associating the process that represents the object with a dispatcher process that will send the next available event only after receiveiving an acknowledgement from the top-level state, indicating that the event has been fully processed. The sending of events to other objects is encapsulated within the exec function, which sends those messages directly through the statechart's *out* port. The treatment of synchronous vs. asynchronous messages is also the responsibility of this function.

Our encoding highlights the relative complexity of the semantics of and-states compared to or-states. This could be used as an argument against and-states as a mechanism to achieve concurrency and perhaps in favour of alternative approaches such as the one taken by the UML-RT profile, where concurrent processes are described as separate active objects with well-defined interfaces whose behaviour is given by simplified State Machines with only or-states.

References

[1] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In *Abstract State Machines – Theory and Applications. International Workshop, ASM 2000, Proceedings.*, volume 1912 of *Lecture Notes in Computer Science*, pages 167 – 186. Springer, 2000.

[2] E. Börger and R. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.

[3] S. Borland. Transforming Statecharts to DEVS. M.Sc. thesis, School of Computer Science – McGill University, 2004.

[4] S. Borland and H. Vangheluwe. Transforming Statecharts to DEVS. In *Proceedings of the 2003 Summer Computer Simulation Conference (SCSC'03) – Student Workshop.*, 2003.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV2: an open source tool for symbolic model checking. Technical report, January 01 2002.

[6] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, 1992.

[7] A. Gawanmeh, S. Tahar, and K. Winter. Formal verification of ASMs using MDGs. *Journal of Systems Architecture: the EUROMICRO Journal*, 52(1–2):15–34, January 2008.

[8] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), 1996.

[9] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[10] V. S. W. Lam. *A Formal Execution Semantics and Rigorous Analytical Approach for Communicating UML Statechart Diagrams.* Ph.D. thesis, University of Bath, 2006.

[11] V. S. W. Lam and J. A. Padget. Formalization of UML statechart diagrams in the π-calculus. In *Australian Software Engineering Conference*, pages 213–223. IEEE Computer Society, 2001.

[12] V. S. W Lam and J. A. Padget. On execution semantics of UML statechart diagrams using the π-calculus. In Ban Al-Ani, Hamid R. Arabnia, and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice, SERP '03, June 23 - 26, 2003, Las Vegas, Nevada, USA, Volume 2*, pages 877–882. CSREA Press, 2003.

[13] K. L. Mcmillan. The SMV system, November 06 1992.

[14] R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and 86, Computer Science Dept., University of Edinburgh, March 1989.

[16] U. Nestmann and B. C. Pierce. Decoding choice encodings. *Information and Computation*, 163(1):1–59, 2000.

[17] M. Y. Ng and M. Butler. Towards Formalizing UML State Diagrams in CSP. In *Proceedings of the 1st International Conference on Software Engineering and Formal Methods SEFM'03*, pages 138–147. IEEE Computer Society, 2003.

[18] Object Management Group. Semantics of a Foundational Subset for Executable UML Models. Request For Proposal. http://www.omg.org/docs/ad/05-04-02.pdf, 2 April 2005.

[19] Object Management Group. Semantics of a Foundational Subset for Executable UML Models. http://www.omg.org/docs/ad/08-05-02.pdf, 2008.

[20] Object Management Group. UML Superstructure Specification v2.1.2. http://www.omg.org/docs/formal/07-11-01.pdf, 2008.

[21] G. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Dept. of Computer Science, Aarhus University, 1981.

[22] E. Posse. *Modelling and simulation of dynamic structure discrete-event systems.* Ph.D. thesis, School of Computer Science – McGill University, October 2008.

[23] E. Posse and H. Vangheluwe. kiltera: A simulation language for timed, dynamic structure systems. In *Proceedings of the 40th Annual Simulation Symposium*, 2007.

[24] W. A. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, 1998.

[25] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* John Wiley & Sons, Ltd., 2000.

[26] S. Van Langenhove. *Towards the Correctness of Software Behavior in UML.* Ph.D. thesis, Universiteit Gent, May 2006.

[27] B. Victor and F. Moller. The Mobility Workbench: A Tool for the Pi-Calculus. Technical Report ECS-LFCS-94-285, School of Informatics – University of Edinburgh, February 1994.

[28] M. von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1(2):130–141, 2002.

[29] W. L. Yeung, K.R.P.H. Leung, J. Wang, and W. Dong. Improvements Towards Formalizing UML State Diagrams in CSP. In *Proc. of the 12th Asia-Pacific Software Engineering Conference, 2005, APSEC '05*, pages 176 – 184. IEEE Computer Society, 2005.

[30] B. P. Zeigler. *Multifacetted modelling and discrete event simulation.* Academic Press, 1984.

[31] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, first edition, 1976.

[32] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, second edition, 2000.