Softw Syst Model
DOI 10.1007/s10270-014-0399-z

REGULAR PAPER

An executable formal semantics for UML-RT

Ernesto Posse · Juergen Dingel

Received: 15 August 2012 / Revised: 20 December 2013 / Accepted: 24 January 2014

Abstract We propose a formal semantics for UML-RT, a UML profile for real-time and embedded systems. The formal semantics is given by mapping UML-RT models into a language called kiltera, a real-time extension of the π-calculus. Previous attempts to formalize the semantics of UML-RT have fallen short by considering only a very small subset of the language and providing fundamentally incomplete semantics based on incorrect assumptions, such as a one-to-one correspondence between "capsules" and threads. Our semantics is novel in several ways: (1) it deals with both state machine diagrams and capsule diagrams; (2) it deals with aspects of UML-RT that have not been formalized before, such as thread allocation, service provision points, and service access points; (3) it supports an action language; and (4) the translation has been implemented in the form of a transformation from UML-RT models created with IBM's RSA-RTE tool, into kiltera code. To our knowledge, this is the most comprehensive formal semantics for UML-RT to date.

Keywords UML-RT · RTE · Modelling · Semantics

Contents

Communicated by Dr. Kevin Lano.

E. Posse (✉) · J. Dingel
Modelling and Analysis in Software Engineering, School of Computing, Queen's University, Kingston, ON, Canada
e-mail: eposse@cs.queensu.ca

1 Introduction

Real-time embedded systems (RTE) have become pervasive in avionics, telecommunications, manufacturing, traffic control, medical devices, and automotive systems. Examples

Published online: 20 February 2014

include antilock breaking systems in modern cars, telecom switches, aircraft flight-control systems, and MRI machines.

RTE systems are typically characterized by being task-oriented, being part of a larger system, and often including safety-critical components. Being a part of a larger system implies that interaction is an essential aspect of an RTE system's behavior. Furthermore, some of the components with which an RTE system interacts are physical components. This entails real-time constraints on the system's behavior.

The term "real-time systems" encompasses a wide range of systems, languages, formalisms, and even definitions of "real-time:"

"There is no general consensus about the definition of both real-time and embedded terms." [41]

"[...] the term "real-time" covers a surprisingly diverse spectrum of systems, ranging from purely time-driven to purely event-driven systems, and from soft real-time systems to hard real-time systems." [54]

In this paper, we focus on the event-driven, soft real-time side of the spectrum, in which it is not critical to meet every deadline every time. While timeliness is the defining characteristic of real-time systems, it is not the only concern. System architecture and good behavior (e.g., safety, liveness, and fairness) are fundamental concerns as well.

1.1 Model-driven development of RTE systems

The development of RTE systems consists of, broadly speaking, several major activities: design and modelling, reasoning and analysis, and implementation and deployment. In design and modelling, engineers rely on languages and formalisms that provide adequate facilities to describe the systems to build, at the desired level of abstraction. Reasoning and analysis reduce risk by providing assurance that the system will meet its requirements. Implementation and deployment realize the system on specific platforms. These activities may be supported by automated tools. For instance, implementation may be aided by automatic code generation and analysis by automatic model (formal) verification.

Model-driven development (MDD) is an approach to software development where models of a system and its components are the main artifact of the development process. In the MDD approach, models provide the basis for the activities outlined above. The engineer designs models, from which an implementation may be automatically generated. The MDD approach also facilitates analysis, as models, by definition, are abstractions of a system, and therefore likely to be simpler and easier to analyze than the final product.

An influential modelling language for the design of RTE systems, which targets event-driven, soft real-time systems, is the Real-Time Object Oriented Modeling language (ROOM) [56]. This language was later made into a UML profile called "UML-RT," introduced in [54,57]. UML-RT is an industrial-strength language, which has enjoyed considerable success with hundreds of large-scale industrial projects and with users in a variety of sectors such as automotive, avionics, and telecommunications, for which ROOM was originally designed. It has been supported by a number of commercial tools, including ObjectTime, Rational Rose RT, IBM's Rational Rose Technical Developer toolkit [22], and IBM Rational Software Architect RealTime Edition (IBM RSA-RTE) [23].

In order to be able to analyze models in a language, the language must have a well-defined semantics; otherwise, the meaning of models would be ambiguous and the analysis results would be ad hoc, applicable only to, for example, specific models or specific implementations. Unfortunately, the semantics of UML-RT has only been defined informally. There have been some attempts at formalizing the semantics of UML-RT (e.g., [4,5,7,16,31,61]), but all of these attempts consider only limited subsets of the language, thus limiting the potential for the analysis.

Our goal is to provide a comprehensive formal semantics for UML-RT, which cannot only serve as a reference semantics but also supports both the execution and the analysis of models.

1.2 UML-based modelling of RTE systems

UML-RT, along with SDL [28] and Acme [17], heavily influenced the development of UML 2 [42,44].

In addition to UML-RT, other UML profiles have been developed to account for timeliness and platform-dependent issues, including the UML Schedulability, Performance and Time profile or UML SPTP [40] and its successor, the UML profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [41], both of which are OMG standards. Other unrelated UML profiles that are not OMG standards have been proposed as well, such as "Real-time UML," a.k.a. RT-UML [11], and variants such as those in [3] and [38].

While there is some overlapping between UML-RT and UML SPTP and MARTE, there are significant differences. At the top level, MARTE consists of three packages: the core package, the design modelling package, and the analysis modelling package. The core package provides constructs to describe non-functional properties, time, generic resource modelling, and allocation modelling. The design modelling package provides facilities for generic components, high-level application modelling, and detailed resource modelling. The analysis package provides facilities for generic quantitative analysis modelling, schedulability analysis, and performance analysis. By contrast, UML-RT focuses on system architecture based on the notion of *capsules* (called *actors* in ROOM), *ports*, *connectors, services* and *capsule structure diagrams*, and event-driven behavior described by *state machines*. MARTE's constructs and models are much more detailed than UML-RT's constructs and models. For instance, in MARTE, the basic unit of concurrent execution is called the RtUnit, defined in the HLAM package. This corresponds roughly to a capsule in UML-RT, but an RtUnit specifies many details such as memory size, message pool policies, and waiting times, whereas a UML-RT capsule abstracts such details.

UML SPTP, MARTE, and other similar profiles target the time-driven and hard real-time side of the spectrum of real-time systems, where the primary concern is timeliness, scheduling, and platform-dependent matters. According to Bran Selic, one of the main authors of ROOM, UML-RT, and UML SPTP,

> "MARTE deals with completely different aspects than UML-RT. MARTE addresses general issues related to real-time systems as they are usually implemented. Thus, it provides facilities for modeling time, resources, real-time operating systems (with their complex schedulers, lightweight threading systems, mutual exclusion facilities, etc.). It also provides support for modeling platforms of all kinds (including hardware), as well as facilities for doing real-time analyses, such as schedulability and queuing network analysis. ROOM/UML-RT, on the other hand, is a level of abstraction above that and does not deal with any of that." [55]

Hence, while MARTE is an appropriate formalism for the design and analysis of time-driven, resource-intensive, and platform-dependent, hard real-time systems, UML-RT may be better suited for soft real-time systems where such level of detail is a secondary concern.

Why should we be concerned with defining a semantics for UML-RT considering that the MARTE profile is available? We believe there are several important reasons: (1) UML-RT is in active industrial use, but there are no UML-RT development tools that provide formal analysis; a formal semantics would provide the basis for such analysis capabilities. (2) While the MARTE profile targets RTE systems, as argued above, UML-RT deals with related but different concerns and level of abstraction. (3) An executable formal semantics can also provide an analysis tool for the simulation and validation of implementations. (4) Given that UML-RT played a central role in the definition of UML 2 and is fully aligned with it, a UML-RT formal semantics can help clarifying issues regarding the formal semantics of UML 2. (5) UML-RT has much in common with several architecture description languages such as AADL [53] and SysML [43], as well as a number of hardware description languages such as VHDL [25], Verilog [24], SystemVerilog [27], SystemC [26] and GDL [21], and a formal semantics for UML-RT can also suggest formal semantics for these languages, or can help elucidating their differences. (6) Formal semantics for foundational languages and calculi abound, but formal semantics for large, complex, industrial-strength languages are few. Our proposal can serve as a showcase for what such semantics can look like.

1.3 UML-RT semantics by translation

There are many approaches to formal semantics, such as denotational, operational, axiomatic, etc. Many such approaches do not yield an executable semantics. Even with operational semantics, which often takes the form of defining some form of transition system, considerable effort is required to obtain an executable artifact. The alternative is to define semantics by translating models to a language that already has a well-defined executable formal semantics.

In this paper, we follow this approach. There are many possible choices for the target language such as CSP or the π-calculus, and we find such examples in the literature (see Sect. 6). However, these alternatives face many difficulties when formalizing a large, complex, high-level language such as UML-RT. Foundational calculi provide a solid basis for a formal semantics, but are limited in that the abstraction gap is often too wide. For example, these low-level calculi often lack higher-level constructs to define complex data types. Without such facilities in place, the task of defining a comprehensive formal semantics of a language such as UML-RT is almost unsurmountable. Hence, we need a target language, which is both executable and formal, as well as having higher-level constructs to make the translation practical.

The target language we have chosen is called kiltera [46, 49–51]. The main reasons for this choice are as follows:

1. kiltera's semantic concepts have many similarities with those of UML-RT, providing a natural representation of UML-RT concepts,
2. kiltera has a well-defined formal semantics based on a real-time extension of the π-calculus [35], a process algebra for modelling and reasoning about concurrent, mobile systems, thus resting on a rich theory that provides a solid foundation for the analysis, and
3. kiltera is a real high-level language with features to ease development and with a working implementation, thus providing the capability of executing models.

The goal of this article is to formally specify a translation from UML-RT models into kiltera. More concretely, we define a map $\mathcal{M}[\![\cdot]\!] : \mathbf{UMLRTC} \rightarrow \mathbf{KLT}$ from UML-RT models to kiltera process terms, where **UMLRTC** is the set of valid UML-RT models and **KLT** is the set of valid kiltera terms.

UML-RT models describe both structural and behavioral aspects of a system. In the structural view, a model consists of a collection of interconnected components called *capsules*, which may have a behavior and may themselves contain *sub-capsules*. The behavior of capsules is specified by *state machines*. In this article, we break down the translation into the behavioral and the structural parts. This is we define a mapping for state machines and a mapping for capsule diagrams. Due to the modular nature of both UML-RT and kiltera, the two mappings are largely independent; therefore, we only need to invoke the state machine translation without reference to its internals, in the capsule diagram mapping. This, in turn, allows for experimentation with alternative semantics, as it opens up the possibility of replacing state machines with some other formalism to specify behavior.

1.4 Shortcomings of existing UML-RT semantics

As we mentioned above, there have been some attempts at formalizing the semantics of UML-RT (e.g., [4,5,7,16,31, 61], and also see Sect. 6).

The existing approaches to formalizing UML-RT fall short not only because of their limited scope and lack of support for syntactic features, but also because they provide fundamentally incomplete semantics. Those approaches will not include behaviors that are possible and, in some cases necessary, of UML-RT models. As a consequence, analysis of system behavior may be incomplete or even erroneous. For example, as we will illustrate later, existing semantics are unable to distinguish between certain fair and unfair behaviors. The reason for this situation is that the existing approaches rely on incorrect assumptions.

Two of these fundamental aspects of the semantics of UML-RT which, with the exception of [31], have been ignored by every other attempt to formalize the language are the relationship between capsules and threads and the mechanism for communication between capsules. Apart from [31], all previous approaches make the incorrect assumptions that each capsule is executed as an independent (concurrent) thread, and that communication between them is direct, thus relying on the communication mechanism of the formalism or language used to describe the semantics (e.g., CSP or LOTOS), or assuming specific message-passing policies (e.g., synchronous communication). But this is not the case: capsules can be assigned to the same thread, sharing the same event queue, and the basic mechanism for message delivery is asynchronous and handled by a *controller* process. This however is not a mere implementation issue or optimization issue, for it is semantically meaningful: different thread assignments and different delivery mechanisms can yield different behaviors, *for the same* UML-RT model. Hence, by ignoring these aspects, other approaches provide an incorrect semantics, which can result in incorrect analysis of system behavior. In this paper, we address this specific issue by providing explicitly in our proposed semantics the controller's role and the assignment of capsules to threads.

As stated above, our goal is to obtain a *comprehensive* account of the UML-RT semantics. While this article fails to cover all elements of UML-RT in detail, we believe it goes well beyond previous attempts to do so, and given its extensible nature, we are increasing the coverage of UML-RT's many features.

1.5 Correctness and validation

One of the main questions regarding the definition of a formal semantics for a language is validation. How do we know that our semantics is correct? Since the language we are formalizing lacks a formal semantics, we cannot prove mathematically the correctness of our semantics. How, then, can we be assured of our semantics' validity? We have addressed this problem by (1) careful study of existing documentation on UML-RT and ROOM, (2) experimentation with the de facto reference implementations of UML-RT, specifically with Rational RoseRT and IBM's RSA-RTE, (3) inspection of code generated by these tools and their runtime systems, (4) consulting with Bran Selic, one of the lead designers of ROOM and UML-RT, and (5) developing a full implementation of the mapping using IBM's RSA transformation tool. The implementation produces code, which can be executed with kiltera's simulator, allowing for validation against the output produced by RoseRT and RSA-RTE. Having an actual implementation of the semantics also differentiates our work from previous attempts.

Just like UML, UML-RT has several *semantic variation points*, where, intentionally or unintentionally, the precise semantics is unspecified. Our definition is intended to be as close as possible to UML-RT as implemented by IBM's RSA-RTE. Nevertheless, some aspects can be considered to be implementation-specific and not mandatory for UML-RT models. In this paper, we will mark such semantic variation points as SVP # and the alternatives are proposed in the "Appendix."

Paper organization This article is organized as follows: In Sect. 2, we present background on UML-RT and kiltera. In Sect. 3, we present a motivating example that shows how thread allocation is essential to the semantics of UML-RT. Section 4 deals with state machines. In Sect. 4.1, we present a formal syntax for state machines and the translation into kiltera is presented in Sect. 4.2. Section 5 addresses capsule diagrams. In Sect. 5.1, we present a formal syntax for capsule diagrams and their translation is presented in Sect. 5.2. In Sect. 6, we discuss some related work and finally Sect. 7 concludes.

In the presentation of the formal translation of both state machines (Sect. 4.2) and capsule diagrams (Sect. 5.2), for each formal definition, we proceed by first informally providing an overview of the concept being defined and then we present the actual formal definition followed by a detailed explanation of the definition and, in the most important cases, an illustrative example.

2 Background

2.1 UML-RT

In this section, we describe informally the main concepts of UML-RT, in particular we describe the notions of capsules and structure diagrams in Sect. 2.1.1 and State Machines in Sect. 2.1.2. While UML-RT covers other types of UML diagrams, we focus only on these two, as they are the most important for UML-RT modelling. For more information on UML-RT, we refer the reader to [54,56,57]. The official account of the UML can be found in [42,44].

2.1.1 Structure diagrams: capsules

UML-RT allows modelling a system's structure through *structure diagrams*, also called *capsule diagrams*. Figure 1 shows a typical UML-RT capsule diagram.

A *capsule*, as its name suggests, is a highly encapsulated active entity, which may have some behavior specified via a state machine (see Sect. 2.1.2). Capsules may execute concurrently with other capsules and communicate with them only by sending and receiving signals through *ports* (p_1, p_2, ..., p_{13} in Fig. 1). Ports in different capsules are linked by *connectors* (labelled l_1, l_2, ..., l_5 in Fig. 1). A connector links only two ports. Each port has a type specified with a *protocol*, which identifies signals that can be sent or received via the port. Communication may be asynchronous or synchronous. 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 *output (send) signals* of one port must be the *input (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, e.g., p_6 and p_9 in Fig. 1. A port marked with $\sim$ implements the conjugated version of a protocol, with the input and output signals inverted.

The set of ports of a capsule defines its *interface*. There are three kinds of ports: *external end ports*, *external relay ports*, and *internal ports*. External end ports are ports linked to external capsules and used directly by the capsule's state machine (if it has one) to either send or receive messages (e.g., port p_2 in Fig. 1). External relay ports are ports directly connected to some sub-capsule (thus, relay messages between some external capsule and some sub-capsule, e.g., ports p_1 and p_3 in Fig. 1). Internal or protected ports are used to communicate between the capsule's state machine and some sub-capsule (e.g., port p_4 in Fig. 1).

Some ports such as p_{12} and p_{13} may be declared as *unwired*, but they may become connected or *wired* at run time by explicit actions on the part of the capsules that own these ports. This is achieved when one of the ports is registered at run time by its capsule as a *service provision point* or SPP for short, under a unique service name, and the other port is registered by its capsule as a *service access point* or SAP for short, under the same unique service name.

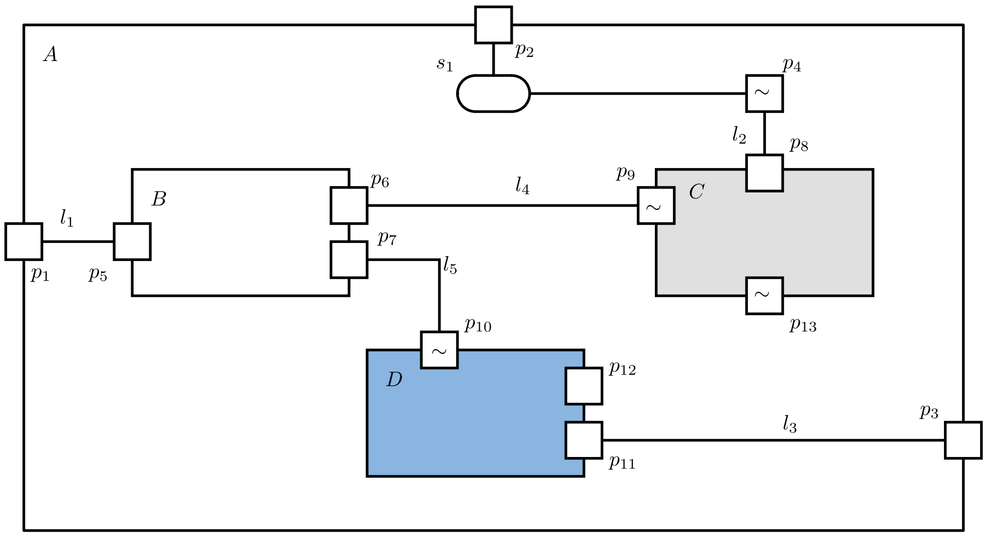

Fig. 1 A UML-RT structure (capsule) diagram

When both ports are registered (which may be done asynchronously), a new connector links them. It is also possible to deregister ports and reregistering them, thus allowing a dynamic reconfiguration of the connections among capsules. SPPs and SAPs were not originally intended by the designers of UML-RT to be used for dynamic wiring between *peer* capsules [55] such as *C* and *D* in Fig. 1. SPPs and SAPs were intended to be used as a mechanism for capsules to access services in the underlying layer or platform, in a multilayer architecture. Nevertheless, the language does not prohibit the dynamic wiring within the same layer, among peer capsules. Furthermore, it is useful for modelling certain kinds of structural changes. For this reason, our mapping considers this operation as any other in the language.

A capsule is a class (in the OO sense) of components with ports. A capsule may have *parts*, which are instances of sub-capsules (and are attributes of the capsule's instance). A sub-capsule part may have one of three possible *roles*: *fixed*, *optional*, or *plug-in*. A fixed sub-capsule is created (respectively, destroyed) when its containing capsule is created (respectively, destroyed) and is permanently attached to its containing capsule. An optional sub-capsule may be *incarnated* (i.e., created) or destroyed at a different time. Plug-in capsule roles are "placeholders" for capsules, which can be filled and removed dynamically, and can be shared between different capsules. In Fig. 1, *B* is a fixed capsule; *C* is an optional role, indicated by its light gray color; and *D* is a plug-in role, indicated by its blue color.

Each capsule is assigned to a *logical thread* of control, which in turn is assigned to some *physical thread*. A logical thread represents a conceptual concurrent thread of execution, while the physical thread is the actual runtime processing thread used by the underlying platform, that is, several capsules/logical threads can share the same real system thread. Each physical thread has a *controller*. A controller drives the execution of all capsules (logical threads) within a single physical thread. It contains the *event pool* for all events whose intended receiver is a capsule associated with the physical thread. It enforces *run-to-completion* semantics, this is, that each state machine of each capsule it controls, processes each event fully, one at a time, before processing the next. This ensures that a capsule's state machine cannot be interrupted while processing an event. Sub-capsules need not belong to the same logical or physical thread as their enclosing capsule.

Viewed as a UML profile, the structural elements of UML-RT correspond to UML 2 elements as follows: a capsule is a component, which is a structured classifier, and therefore an encapsulated classifier. Ports and connectors have the same name in UML 2, but in UML-RT it is useful to distinguish between the internal, relay, and end ports. SAPs and SPPs are particular kinds of ports.

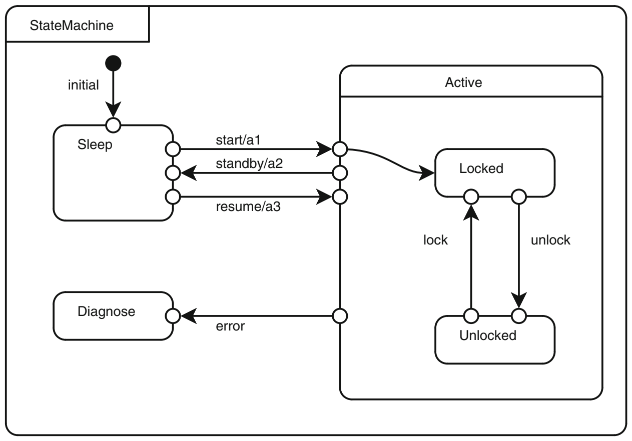

Fig. 2 A UML-RT state machine diagram

2.1.2 Behavior diagrams: state machines

The behavior of a capsule is specified using UML-RT state machines [54,56,57], which are similar to UML state machines [42,44] but with some key differences. Figure 2 shows a typical UML-RT state machine diagram.

A UML-RT state machine has hierarchical states and guarded transitions, which are triggered by events 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. 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, and any transition chain enabled will be fully followed and its actions fully executed before the next event is handled.

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* (in UML terminology, *entry* and *exit pseudostates*). 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 form *a transition chain*, which is executed as one step.
3. In UML-RT, entry points are by default connected to deep-history pseudostates. 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 not been previously visited and there is an initial transition pointing to the *default state*, then the initial transition is followed and the default state entered. If, however, n has been visited previously, then the last sub-state visited in n is entered. If it has not been visited and there is no initial transition, no sub-state is entered and the state machine remains "at the border" of n. 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 pseudostates, but to "shallow" history pseudostates, or to the boundary of the state, in which case an initial state is always entered, if an entry point is not explicitly connected to a sub-state. Since all states have deep-history semantics, we avoid the common notation of depicting deep-history pseudostates explicitly, to avoid clutter in the diagrams.
4. Actions may be related to concepts specific to UML-RT such as capsule operations. In particular, an action may send an event through a port, create or destroy an optional capsule, import or deport a plug-in capsule, connect or disconnect unbound ports, and perform normal operations on objects.
5. 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 time-out event from the port which it can process as any other signal.

2.1.3 Time

In UML-RT, time is assumed to progress according to an external timing service (usually provided by the underlying platform). The timing service adheres to a timing protocol with a distinguished *time-out signal* and a period or a deadline. The timing service is accessible by UML-RT models through a standard port with the corresponding timing protocol, so time signals can be treated as any other signal. Since the timing service is external, it can proceed in any way that maintains time consistency, i.e., if two timers with time-out signals tmo_1 and tmo_2 are set up at the same time t_0 with time-outs $t_1 > t_0$ and $t_2 > t_0$, respectively, and such that $t_1 < t_2$, then the timing service must guarantee that signal tmo_1 will be triggered before tmo_2. Besides this requirement, the semantics of UML-RT does not make any assumptions about the rate of progress of these clocks. Furthermore, since UML-RT is not concerned with performance or scheduling, it makes no assumptions about the duration of specific actions. We assume that individual actions in the underlying action language take a negligible amount of time with respect to the minimum time unit of the time services used. Furthermore, other activities such as entering or exiting a state, or relaying a message on a relay port, also take a negligible amount of time. In the case of asynchronous communication between capsules, the amount of time between the sending of a message and its reception and consumption is undetermined. If a capsule is in a state listening to a normal port and a time-out port, and the environment sends the message before the time-out, the language does not guarantee that the message will be consumed before the time-out signal arrives.

2.2 kiltera

Our approach to formalize the semantics of UML-RT is to use a *process calculus* or *process algebra* to describe the behavior of a model. Process calculi or process algebras are mathematical formalisms for modelling and reasoning about concurrent systems in which a broad set of algebraic, logic, and set theoretic techniques can be used to analyze system behavior. Some of the best-known process calculi are CCS [34], CSP [19], ACP [2], and the π-calculus [35].

kiltera [46,49–51] 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 [48], which is a real-time extension of the asynchronous π calculus [6,20,35], one of the best-known variants of the π-calculus.

Just as in the π-calculus, the central notions are those of *process* and *channel*. A π_{klt} term represents a process or set of processes running concurrently. Processes interact by asynchronous message passing over channels. In kiltera , we identify *events* and channels: *triggering* an event *start* is the same as sending a message over a channel named *start*, and *listening* to an event is the same as waiting for input on a channel. This event-oriented terminology is due to the fact that kiltera was originally designed in the context of modelling and simulation of c as treated in [63,64]. Just like the π calculus, kiltera supports channel mobility: the ability to send channels (i.e., events) as part of messages. This allows the topology of the network to change dynamically.

In addition to communication primitives, π_{klt} extends the asynchronous π-calculus by introducing timing constructs (e.g., delaying the execution of a process, recording waiting times, and time-outs), primitive data values and data structures, pattern matching on input, nested process, and function definitions with lexical scoping. These characteristics make it a high-level language, which facilitates our description of the semantics of UML-RT, while still having a formal semantics.

The formal 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

P	$::=$	$\texttt{stop}$	Stopped process
	$\mid$	$\texttt{done}$	Successful termination
	$\mid$	$a!E$	Trigger/Output
	$\mid$	$\texttt{when}\,\{G_1 \to P_1 \mid \cdots \mid G_n \to P_n\}$	Listener/Input
	$\mid$	$\texttt{new}\, a_1, ..., a_n\, \texttt{in}\, P$	New/Hide
	$\mid$	$\texttt{if}\, E\, \texttt{then}\, P_1\, \texttt{else}\, P_2$	Conditional
	$\mid$	$\texttt{wait}\, E \to P$	Delay
	$\mid$	$A(E_1, ..., E_n)$	Instantiation/Call
	$\mid$	$\texttt{def}\,\{D_1; ...; D_n\}\,\texttt{in}\, P$	Local definitions
	$\mid$	$P_1 \parallel P_2$	Parallel composition
	$\mid$	$P_1; P_2$	Sequential composition
	$\mid$	$x := E$	Assignment
G	$::=$	$a?R@y$	Listener/input guard
D	$::=$	$\texttt{proc}\, A(x_1, ..., x_n) = P$	Process definition
	$\mid$	$\texttt{func}\, f(x_1, ..., x_n) = E$	Function definition
	$\mid$	$\texttt{var}\, x = E$	Variable definition
E	$::=$	$\texttt{null} \mid r \mid \texttt{true} \mid \texttt{false} \mid \text{“}s\text{”} \mid x$	
	$\mid$	$\langle E_1, ..., E_m\rangle \mid f(E_1, ..., E_m)$	
R	$::=$	$\texttt{null} \mid r \mid \texttt{true} \mid \texttt{false} \mid \text{“}s\text{”} \mid x$	
	$\mid$	$\langle R_1, ..., R_m\rangle$	

Fig. 3 π_{klt} syntax

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.

We have developed an implementation of the language based on an abstract machine, which has been proven sound with respect to π_{klt}'s operational semantics. The core simulation algorithm consists of event scheduling as known in discrete-event simulation [64]. 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. Our interpreter is a prototype implemented in Python and does not use a real-time operating system; thus, in real-time mode, timing constraints are only approximated.

The full language also includes some constructs for distributed computing, allowing the execution of processes in logical *sites*. The simulator allows assigning kiltera sites to different physical machines, and distributed simulation is performed using a variation of the Time-Warp algorithm [29]. We have used kiltera in the modelling of complex systems such as automobile traffic simulation and cloud computing environments. kiltera has been used for teaching in graduate courses at McGill and Queen's universities. Our kiltera simulator is available for download at http://www.kiltera.org.

2.2.1 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 of all π_{klt} ***process terms***, denoted **KLT**, is defined by the BNF in Fig. 3. The same BNF defines the set **Expr** of ***expressions***, ranged over by $E, E', \ldots$; the set **Patts** of ***patterns***, ranged over by $R, R', \ldots$; and the set **Defs** of ***definitions***, ranged over by $D, D', \ldots$. We usually write $a, b, c \ldots$ for channel/event names, $A, B, C, \ldots$ for process names, $x, y, z, \ldots$ for variables.

2.2.2 Informal semantics

We now describe informally the language's semantics. For a formal semantics of the language, see [48]. For earlier versions of the semantics, see [46,47,50].

- Expressions E are either constants (`null` represents the *null* constant), variables (x), tuples of the form $\langle E_1, \ldots, E_m \rangle$ or function applications $f(E_1, \ldots, E_m)$. Patterns R have the same syntax as expressions, except that they do not include function applications.
- The term `stop` represents the stopped process: it has no actions.
- The process `done` represents successful termination.
- The process $a!E$ is a *trigger*; it triggers an event a with the value of E. Alternatively, we can say that it sends the value of E over a channel a. This is an asynchronous message sending, with no specific buffering policy mandated by the semantics. The expression E is optional: $a!$ is shorthand for $a!\texttt{null}$.
- A process $\texttt{when}\,\{G_1 \rightarrow P_1 \mid \cdots \mid G_n \rightarrow P_n\}$ is a *listener*. Each G_i is a *guard* of the form $a_i?R_i\,@\,y_i$ where a_i is an event/channel name R_i is a *pattern*, and y_i is an optional variable. This process listens to all channels (or events) a_i, and when a_i is triggered with a value V that matches the pattern R_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. 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 $\langle 3, \texttt{true}, 7\rangle$ matches the pattern $\langle 3, x, y\rangle$ with the resulting binding $\{\texttt{true}/x, 7/y\}$. The scope of these bindings is the corresponding P_i. The suffixes R_i and $@\,y_i$ are optional: $a? \rightarrow P$ is equivalent to $a?x\,@\,y \rightarrow P$ for some fresh names x and y.
- The process $\texttt{new}\,a_1, \ldots, a_n\,\texttt{in}\,P$ hides the names a_i from the environment, so that they are private to P. Alternatively, $\texttt{new}\,a_1, \ldots, a_n\,\texttt{in}\,P$ can be seen as the creation of new names, i.e., new events or channels, whose scope is P.
- The process $\texttt{wait}\,E \rightarrow P$ is a *delay*: it delays the execution of process P by an amount of time equal to the value of the expression E. The value of E is expected to be a nonnegative real number. If the value of E is negative, $\texttt{wait}\,E \rightarrow P$ cannot perform any action. Similarly, terms with undefined values (e.g., $\texttt{wait}\,1/0 \rightarrow P$) or with incorrectly typed expressions (e.g., $\texttt{wait true} \rightarrow P$) cause the process to stop. Since the language is untyped, we do not enforce these constraints statically.
- The process $\texttt{if}\,E\,\texttt{then}\,P_1\,\texttt{else}\,P_2$ is a conditional with the standard meaning. $\texttt{if}\,E\,\texttt{then}\,P$ is shorthand for $\texttt{if}\,E\,\texttt{then}\,P\,\texttt{else done}$.
- The process $P_1 \parallel P_2$ is the parallel composition of P_1 and P_2. We also allow an *indexed parallel composition*, written $\prod_{i \in I} P_i$ to stand for $P_1 \parallel P_2 \parallel \cdots \parallel P_n$ for some index set $I = \{1, 2, \ldots, n\}$.
- The term $P_1; P_2$ is the sequential composition of P_1 and P_2.
- The term $\texttt{def}\,\{D_1; \ldots; D_n\}\,\texttt{in}\,P$ 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). Each D_i can be either a *process definition* $\texttt{proc}\,A(x_1, \ldots, x_n) = P$, a *function definition* $\texttt{func}\,f(x_1, \ldots, x_n) = E$ or a *local variable definition* $\texttt{var}\,x = E$.
- The term $x := E$ assigns the value of E to the local variable x.
- The process $A(E_1, \ldots, E_n)$ creates a new instance of a process defined by $\texttt{proc}\,A(x_1, \ldots, x_n) = P$, defined in some enclosing scope, where the ports or parameters $x_1, \ldots, x_n$ are substituted in the body P by the values of expressions $E_1, \ldots, E_n$, which may be channel names.

2.2.3 Some examples and usage patterns

In order to give the reader some intuition about the semantics of π_{klt}, we present some representative examples and common patterns.

Interaction The process $a! \parallel \texttt{when}\,\{a? \rightarrow P\}$ results in one interaction between the processes and then continues as $\texttt{done} \parallel P$, which is the same as just P.

Choice The term $a! \parallel \texttt{when}\,\{a? \rightarrow P | b? \rightarrow Q\}$ reduces to P, while $b! \parallel \texttt{when}\,\{a? \rightarrow P | b? \rightarrow Q\}$ reduces to Q. If the environment of a listener triggers more than one of the listener's guards, the choice is non-deterministic: $a! \parallel b! \parallel \texttt{when}\,\{a? \rightarrow P | b? \rightarrow Q\}$ can reduce to either $b! \parallel P$ or $a! \parallel Q$.

Pattern-matching For interaction to happen, data received must match the expected pattern: the process $a!$"`hi`" $\parallel$ $\texttt{when}\,\{a?$"`hi`" $\rightarrow P\}$ reduces to P. On the other hand, $a!$"`hi`" $\parallel \texttt{when}\,\{a?$"`hey`" $\rightarrow P\}$ does not result in an interaction because the data sent over a ("`hi`") do not match the expected pattern ("`hey`"). Hence, the two processes remain the same. If the pattern has variables, a successful communication results in substituting the corresponding variables by the received values: $a!$"`hi`" $\parallel \texttt{when}\,\{a?x \rightarrow P\}$ results in $P\{$"`hi`"$/x\}$, this is, substituting every free occurrence of x in P by "`hi`". The same holds for more complicated patterns: the term $a!\langle$"`hi`"$, 6\rangle \parallel \texttt{when}\,\{a?\langle$"`hi`"$, x\rangle \rightarrow P\}$ results in $P\{6/x\}$.

Local channels The `new` construct introduces new names and restricts their scope. For example, in the term $a!1 \parallel \texttt{new}\,a\,\texttt{in}\,(a!2 \parallel \texttt{when}\,\{a?x \rightarrow P\})$, the a in $a!1$ is different from the one in $a!2$. The whole term reduces to $a!1 \parallel P\{2/x\}$.

Barriers and joining It is common for a process to wait for several other processes before continuing. This can be achieved with nested listeners: in $(\texttt{wait}\,3 \rightarrow a!) \parallel b! \parallel \texttt{when}\,\{a? \rightarrow \texttt{when}\,\{b? \rightarrow P\}\}$, process P will begin only when both a and b have been triggered. This example also shows that the triggers are *persistent*, that is, the trigger $b!$ is not lost if no other process is listening to b, and remains available until some process is ready to accept it. So the whole process waits 3 time units and becomes $a! \parallel b! \parallel \texttt{when}\,\{a? \rightarrow \texttt{when}\,\{b? \rightarrow P\}\}$, which then becomes $b! \parallel \texttt{when}\,\{b? \rightarrow P\}$ which finally becomes P. This notion of nested listeners as barriers is so useful that we will write $\texttt{when}\,\{\langle a, b\rangle? \rightarrow P\}$ as syntactic sugar for $\texttt{when}\,\{a? \rightarrow \texttt{when}\,\{b? \rightarrow P\}\}$. The sequential composition operator is also useful for joining processes: in $(P \parallel Q); R$ process R will start only after both P and Q are done.

Process definitions Process definitions allow us to encapsulate processes, giving them a specific interface and be reused in the scope of their definition. For example, $\texttt{def}\,\{\texttt{proc}\,P(x) = x!;\ \texttt{proc}\,C(y) = \texttt{when}\,\{y? \rightarrow Q\}\,\}\,\texttt{in}\,\texttt{new}\,a\,\texttt{in}\,(P(a) \parallel C(a))$ results in the same process as the term $\texttt{new}\,a\,\texttt{in}\,(a! \parallel \texttt{when}\,\{a? \rightarrow Q\})$. The parameters of a process definition can be thought of as its interface, its ports, so when we invoke the process definition, we can visualize it as creating an instance of the process and "hooking up" channels to its ports, e.g., in $P(a)$, we are instantiating P and hooking up the local channel a to the new instance's port x. Nevertheless, parameters are not required to be only channels or events, but they can be any value. This fact is used for example to keep track of additional state variables.

Recursion The body of a process can refer to itself, or even to other processes in the same definition group (or any enclosing process definitions). Recursion is used by a process to keep itself alive and possibly change its connections by invoking itself with different parameters. For example, consider the definition $\texttt{proc}\,A(x, y) = \texttt{when}\,\{x?z \rightarrow (y! \parallel A(z, y))\}$. Then, executing $A(a, b) \parallel a!c$ will result in $\texttt{when}\,\{a?z \rightarrow (b!\} \parallel A(z, b)) \parallel a!c$ which will then reduce to $b! \parallel A(c, b)$.

Lexical scoping This applies to names introduced with $\texttt{new}$, names introduced with $\texttt{def}$ and pattern variables. This is the occurrence of a name x always refers to the closest enclosing construct that declares it, e.g., in $\texttt{proc}\,A(x, y, z) = \texttt{when}\,\{x?y \rightarrow \texttt{new}\,z\,\texttt{in}\,y!\langle x, z\rangle\}$, in the innermost term $y!\langle x, z\rangle$, x refers to the first parameter of A, y refers to the pattern in the listener's guard $x?y$ (not A's second parameter), and z refers to the one introduced by $\texttt{new}\,z$ (and not to A's third parameter).

Channel mobility Channels or events are first-class objects, so they can be included in messages: reducing $a!b \parallel \texttt{when}\,\{a?x \rightarrow x!c\}$ results in $b!c$. This is allowed even for private or local names. For example, the term $\texttt{when}\,\{a?x \rightarrow x!c\} \parallel \texttt{new}\,b\,\texttt{in}\,(a!b \parallel P)$ reduces to the term $\texttt{new}\,b\,\texttt{in}\,(b!c \parallel P)$. In this case, the right-hand sub-process sent a private channel b to the left-hand sub-process via a. Hence, the left-hand process evolves into $b!c$ becoming aware of the private b.[1]

Asynchronous message passing As in the asynchronous π-calculus, asynchronous communication is modelled by syntactically restricting the output operator by not allowing it to have a continuation. In practice, however, it is often desired to allow writing, e.g., $a!1 \rightarrow P$. This however is only syntactic sugar for $a!1 \parallel P$, as the process P is free to continue without having to wait for the output $a!1$ to be consummated.

Message acknowledgment and response Since communication is asynchronous, when sending a message, the sender does not wait for the receiver to get and acknowledge the message, e.g., in $a!$"hi"; Q process Q can begin before any process receives the message sent over a. Nevertheless, we often wish to receive an acknowledgment or response from a receiver. A common way to do this in the π-calculus is to use channel mobility: create a local channel, say r where the sender will expect the acknowledgment or response, send r as part of the query and listen to r before proceeding. The response message on r may be empty to signal acknowledgment, or may include data, such as the answer to the query. This can also be seen as a simple way to encode synchronous message passing or remote procedure calls. The response channel needs to be local to remain private, avoiding interference from other processes. For example, the sender could be $\texttt{proc}\,S(q) = \texttt{new}\,r\,\texttt{in}\,(q!r \parallel \texttt{when}\,\{r?x \rightarrow P\})$ and the receiver could be $\texttt{proc}\,R(q) = \texttt{when}\,\{q?r \rightarrow (Q; r!\text{"}\texttt{result}\text{"})\}$. Thus, the sender sends a query on channel q including its private channel r where it will expect the response and then listens to r. Once the response arrives, it proceeds as P. The receiver waits for a query on q and when the query arrives, it is expected to come with a response channel r. Then, it proceeds to do some task Q and when it is done, it sends the result on channel r. We use this pattern repeatedly throughout our translation.

Process names as parameters In process definitions, process invocations, expressions, and patterns, we allow the names x to be process and function names as well. This is an essential feature that allows us to write generic processes, for example: $\texttt{def}\,\{\texttt{proc}\,A(x) = x!1;\ \texttt{proc}\,B(y, Z) = Z(y)\,\}\,\texttt{in}\,\texttt{new}\,u\,\texttt{in}\,B(u, A)$. In this example, the second

[1] In the π-calculus literature, this is known as *scope extrusion* as the lexical scope of the private name is effectively extended beyond its original scope.

parameter passed to B is A, so executing $B(u, A)$ results in $A(u)$.

Auxiliary functions While data structures such as lists and dictionaries (associative tables) are not primitive, they can be encoded in this language. It is outside the scope of this paper to provide such encodings, but for convenience we will assume the following functions as primitive:

– `empty_list`: the empty list constant,
– `list_add`(*item*, *list*): returns the list that has *item* as the first element and *list* as the remainder,
– `list_pop`(*list*): returns a pair ⟨*item*, *rem*⟩ where *item* was the first element of *list* and *rem* was the rest,
– `list_del`(*item*, *list*): returns the *list* without *item*,
– `list_isempty`(*list*): returns true if the list is empty, and false otherwise,
– `empty_dict`: the empty dictionary constant,
– `dict_put`(*key*, *value*, *dict*): returns a dictionary that adds the association ⟨*key*, *value*⟩ to the dictionary *dict*, if there was no pair with the given *key*, otherwise it replaces the existing association ⟨*key*, *v*⟩ with the association ⟨*key*, *value*⟩,
– `dict_get`(*key*, *dict*) returns the *value* associated with the *key* in the dictionary *dict*, or `null` if the *key* has no associated value,
– `dict_del`(*key*, *dict*) returns the dictionary *dict* with any association ⟨*key*, *v*⟩ removed.

2.3 Additional preliminaries

Here, we define some additional notation used throughout the paper.

We write $1..k$ for the set $\{1, 2, \ldots, k\}$. Sequences will be enclosed in ⟨ and ⟩. A sequence name will be denoted with an arrow on top, and its elements subscripted with their index, beginning from 1: $\tilde{x} = \langle x_1, x_2, x_3, \ldots\rangle$. A finite sequence $\langle a_1, \ldots, a_k\rangle$ will be abbreviated as $a_{1..k}$. The empty sequence is denoted $\langle\rangle$, or ϵ. We will also use standard set operators for sequences, in particular we write $x \in \tilde{x}$ for the membership of x in the sequence $\tilde{x}$.

3 The significance of thread allocation

As suggested in the introduction, thread allocation is a fundamental aspect of UML-RT, which is overlooked by the existing attempts to formalize its semantics. To illustrate the semantic importance of this issue, we now present an example that highlights how thread allocation affects the semantics. This example also illustrates several of the UML-RT features that our proposed formal semantics addresses.

Example 1 Suppose that some system A uses some sub-component B to perform a task, but B may fail to answer requests timely. For such situations, A includes an optional sub-component C as a fallback. At first, A will attempt to make a request to B, and if B responds, then it will continue to behave in some specified way. But if B has not responded within a certain amount of time, A will send the request to C, while still listening to a possible response from B. If a response from C arrives, the behavior of A will continue in a different way than if the response came from B.

The model is shown in Fig. 4. In this model, we have a top-level capsule A with a fixed sub-capsule B and an optional sub-capsule C. A is connected to B via the l_1 connector, so ports p_1 (internal) of A and p_3 of B are wired. However ports p_2 of A and p_4 of C are unwired. Their behavior is as follows: capsule A registers p_2 as an SAP under some service name “`s`" and incarnates a capsule in C in some logical thread L_1. Then, it sets up a timer to trigger in 1.0 time units and sends an event e_1 to B via p_1. If it receives a response from B on port p_1, it goes to state n_4. After the time-out event is received on port tmo, it sends event e_1 to C via port p_2 and waits for a reply from either B on p_1 or C on p_2, leading to states n_4 or n_5, respectively. Capsule B simply waits for input from A on port p_3 and then responds on the same port with an event e_2. Capsule C is very similar to B, except that in its initial state, it registers its port p_4 as an SPP under the service name “`s`", thus connecting it to A's internal port p_2. Then, it waits for input on p_4 and responds to the same port with an event e_2. All actions in Fig. 4 are assumed to be entry actions.

The standard assumption in the literature is maximum concurrency: each capsule is executed by its own thread. In particular, the logical thread L_1 in which C is incarnated is executed in a separate physical thread. Under such assumption, there are many possible behaviors. Each capsule will have its own event pool, and therefore, transitions and actions can be interleaved in any order, so it is possible that B performs transition t_5 before C performs transition t_6, leading to A ending up in state n_4, but it is also possible that C performs transition t_6 before B performs transition t_5, causing A to end up in state n_5.

However, if C is incarnated in the same physical thread as B (the logical thread L_1 is associated with the same physical thread of B), then the behavior becomes deterministic, as both B and C will share the same controller and therefore the same event pool. Since A always sends a message to B before sending a message to C, it is guaranteed that the message addressed to B will be in the queue before the message addressed to C (assuming the same priorities). Therefore, transition t_5 of B is always triggered first, and the entry action of state n_8 of B is executed as part of the same run-to-completion step. Hence, B sends event e_2 back to A before C does, which implies that transition t_2 or t_3 of A

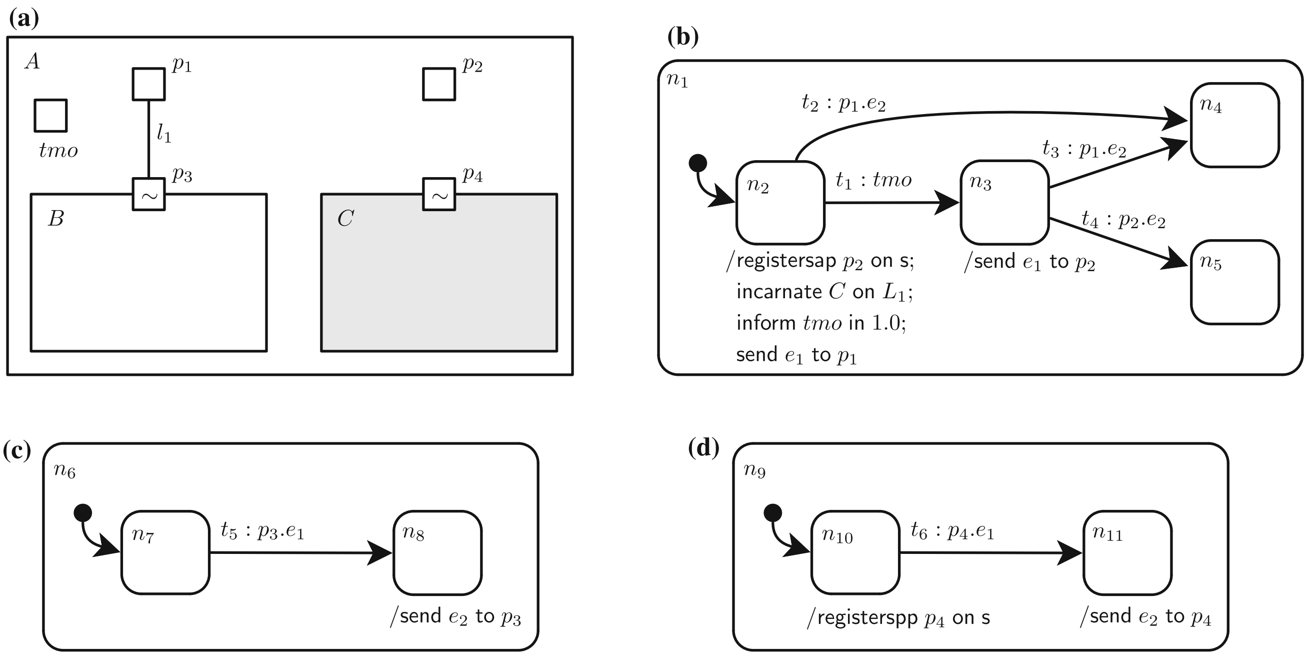

Fig. 4 Model sensitive to thread assignment. **a** Structure diagram. **b** State machine diagram for capsule A. **c** State machine diagram for capsule B. **d** State machine diagram for capsule C

is triggered and A always ends up in state n_4. This implies that under this thread assignment, no execution of the system will lead to A being in state n_5, with B always winning the race over C. In other words, we get only unfair executions.

It is essential to point out that when B and C are assigned to the same thread, the system's behavior is deterministic *regardless of the platform*. This model can be easily extended to show that if B and C are associated with the same thread, then the system will deadlock, whereas if they are associated with different threads, they may not deadlock. For example, we can add another transition from n_8 in B, which listens to a message e_3 from A but this message is sent from state n_5 in A. In such a model, if B and C are associated with the same thread, then A will never end up in state n_5 and therefore B will never get to receive e_3. Similarly, we can modify the example to show that under certain thread assignments, the system is unfair while under others it may be fair. For example, if we add a loop in the form of a transition from n_4 to n_2 in A, and similar loops with transitions from n_8 to n_7 in B and from n_{11} to n_{10} in C with trigger e_4 so that these systems could run forever, and we add an action to n_5 in A that sends e_3 to C, then it becomes evident that under the assignment of B and C to the same thread, C gets stuck in n_{11} so it never gets a chance to continue executing. In other words, the behavior becomes unfair.

It is not hard to make a simple model which, by using synchronous communication, exhibits deadlocks when the sender and receiver are on the same thread, but is deadlock-free when they are assigned to different threads.

By ignoring thread assignment, existing proposals of formal semantics for UML-RT fail to discriminate between a system that will deadlock from a system that may not deadlock and it will fail to discriminate from a fair system and an unfair system.

Thread allocation is important in practice, as it is related to optimize an deployment. The engineer may choose between different allocations depending on available resources or platform constraints. But, as this example shows, naively assuming that thread allocation is only a matter of optimization, or that it is only a platform-dependent issue is misleading at best, and may result in incorrect runtime behaviors or incorrect analysis results at worst. Therefore, a truly useful formal semantics of UML-RT must take it into account. The semantics proposed in this paper accounts for thread assignment and therefore can distinguish between certain systems with respect to some safety, liveness, and fairness properties that other semantics for UML-RT fail to distinguish.

4 State machines

Now, we begin the presentation of our semantics. In this section, we show how to map UML-RT state machines into kiltera. We first introduce a syntax to describe these state machines in Sect. 4.1 and then we describe how to map them into π_{klt} processes in Sect. 4.2.

4.1 A syntax for UML-RT state machines

We use a mathematical notation for state machines, adapted from [60], which allows us to define the mapping compositionally.

In the sequel, we will use the following sets:

- $\mathcal{N}_{states}$: the set of all possible *state names*; we use $n, n_1, n_2, \ldots, m, \ldots$ for elements in $\mathcal{N}_{states}$;
- $\mathcal{N}_{enp}$: the set of all possible *entry point names*; we use $a, a_1, a_2, \ldots$ for elements in $\mathcal{N}_{enp}$;
- $\mathcal{N}_{exp}$: the set of all possible *exit point names*; we use $b, b_1, b_2, \ldots$ for elements in $\mathcal{N}_{exp}$;
- $\mathcal{N}_{cp} \overset{def}{=} \mathcal{N}_{enp} \cup \mathcal{N}_{exp}$: the set of all *connection point names*; we use $c, c_1, c_2, \ldots$ for connection points.
- $\mathcal{N}_{ports}$: the set of all possible *port names*; we use $p, p_1, p_2, \ldots$ for elements in $\mathcal{N}_{ports}$;
- $\mathcal{N}_{evt}$: the set of all possible *event names* including the "non-event" $\perp$, used to mark transitions without a trigger; we use $e, e_1, e_2, \ldots$ for elements in $\mathcal{N}_{evt}$;
- **Trig**: the set of all possible *triggers*: it is defined as $\mathcal{N}_{ports} \times \mathcal{N}_{evt}$. We write $p.e$ for $(p, e) \in \textbf{Trig}$.
- **Vals**: is a set of possible *data values*.
- **Guards**: the set of possible transition *guards* (which are boolean expressions over port names, capsule attributes, and event data). We write $g, g_1, g_2, \ldots$ for guards.
- **Acts**: the set of all possible *actions* including the "non-action" $\perp$, i.e., the action that does nothing; we use $f, f_1, f_2, \ldots$ for transition actions, en for entry actions and ex for exit actions in **Acts**;
- $\mathbb{B} \overset{def}{=} \{\mathsf{false}, \mathsf{true}\}$ 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. 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.
- For every state name $n \in \mathcal{N}_{states}$, there is an entry point name $\mathsf{den}_n \in \mathcal{N}_{enp}$ and an exit point name $\mathsf{dex}_n \in \mathcal{N}_{exp}$. These denote the default entry and exit points of a state, respectively, that 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 does not 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 **Kinds** $= \{\mathsf{in}, \mathsf{out}, \mathsf{sib}\}$ represent the set of transition ***kinds***, (respectively, in for incoming, out for outgoing, and sib for sibling). The set of all possible transitions is $\textbf{Trans} \overset{def}{=} \textbf{Kinds} \times \mathbb{B} \times \mathcal{N}_{cp} \times \mathcal{N}_{cp} \times \textbf{Trig} \times \textbf{Guards} \times \textbf{Acts}$. Given a transition $t = (k, l, c_1, c_2, e, g, f) \in \textbf{Trans}$, we define the following functions:[2]

$\mathsf{kind}(t) \overset{def}{=} k$	The kind of transition
$\mathsf{first}(t) \overset{def}{=} l$	Whether t is the first in a chain
$\mathsf{src}(t) \overset{def}{=} c_1$	The source of the transition
$\mathsf{targ}(t) \overset{def}{=} c_2$	The target of the transition
$\mathsf{trig}(t) \overset{def}{=} e$	The trigger event of the transition
$\mathsf{guard}(t) \overset{def}{=} g$	The guard of the transition
$\mathsf{act}(t) \overset{def}{=} f$	The action of the transition

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:

$s ::= [n, A, B, en, ex]$	Basic state
$\mid\ [n, A, B, S, d, T, en, ex]$	Composite state

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

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

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

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

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

$\mathsf{name}(s) \overset{def}{=} n$	The name of the state s
$\mathsf{entries}(s) \overset{def}{=} A$	The set of entry points s
$\mathsf{exits}(s) \overset{def}{=} B$	The set of exit points of s
$\mathsf{substates}(s) \overset{def}{=} S$	The set of substates of s
$\mathsf{trans}(s) \overset{def}{=} T$	The set of transitions of s
$\mathsf{default}(s) \overset{def}{=} s_d$	The default (initial) sub-state of s
$\mathsf{enact}(s) \overset{def}{=} en$	The set of entry actions of s
$\mathsf{exact}(s) \overset{def}{=} ex$	The set of exit actions of s

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

1. If $\mathsf{first}(t) = \mathsf{false}$ then $\mathsf{trig}(t) = \perp$
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 $\mathsf{targ}(t) \in \mathsf{entries}(s_j)$.
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 $\mathsf{targ}(t) \in \mathsf{entries}(s_i)$.
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 $\mathsf{targ}(t) \in B$.

In the remainder, we will omit the entry and exit actions when $en = \perp$ and $ex = \perp$, and if we omit a transition's guard, it is assumed to be `true`. Also, in our examples, the transition's labels have the general form $t : p.e[g]/a$ where t is the transition's name (only used for readability, but not part of the formal definition), $p.e \in \mathbf{Trig}$ is the transition's trigger with port $p \in \mathcal{N}_{ports}$ and event $e \in \mathcal{N}_{evt}$, $g \in \mathbf{Guards}$ is the transition's guard, and $a \in \mathbf{Acts}$ is the transition's action. All of these items are optional.

Example 2 Consider the state machine shown in Fig. 5. This is encoded in our syntax as follows:

$$\begin{aligned}
s_1 &\overset{def}{=} [n_1, \{\mathsf{den}_{n_1}\}, \{\mathsf{dex}_{n_1}\}, \langle s_2, s_5\rangle, 1, \{t_1, t_2\}] \\
s_2 &\overset{def}{=} [n_2, \{\mathsf{den}_{n_2}, a_1, a_2\}, \{\mathsf{dex}_{n_2}, b_1, b_2\}, \\
&\qquad \langle s_3, s_4\rangle, 1, \{t_3, t_4, t_5, t_6, t_7\}] \\
s_3 &\overset{def}{=} [n_3, \{\mathsf{den}_{n_3}\}, \{\mathsf{dex}_{n_3}\}] \\
s_4 &\overset{def}{=} [n_4, \{\mathsf{den}_{n_4}, a_3\}, \{\mathsf{dex}_{n_4}\}] \\
s_5 &\overset{def}{=} [n_5, \{\mathsf{den}_{n_5}\}, \{\mathsf{dex}_{n_5}\}]
\end{aligned}$$

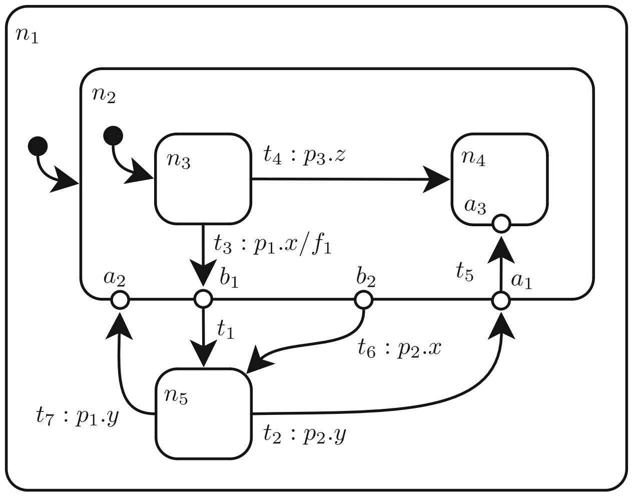

Fig. 5 A state machine with composite states

with transitions

$$\begin{aligned}
t_1 &\overset{def}{=} (\mathsf{sib}, \mathsf{false}, b_1, \mathsf{den}_{n_5}, \perp, \perp) \\
t_2 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_5}, a_1, p_2.y, \perp) \\
t_3 &\overset{def}{=} (\mathsf{out}, \mathsf{true}, \mathsf{dex}_{n_3}, b_1, p_1.x, f_1) \\
t_4 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_3}, \mathsf{den}_{n_4}, p_3.z, \perp) \\
t_5 &\overset{def}{=} (\mathsf{in}, \mathsf{false}, a_1, a_3, \perp, \perp) \\
t_6 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, b_2, \mathsf{den}_{n_5}, p_2.x, \perp) \\
t_7 &\overset{def}{=} (\mathsf{sib}, \mathsf{true}, \mathsf{dex}_{n_5}, a_2, p_1.y, \perp)
\end{aligned}$$

4.2 Translating state machine diagrams

4.2.1 An informal description

State hierarchy 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. Recall that π_{klt} process definitions can be nested with the `def` $\{\ldots\}$ `in` P construct. We make use of this feature, to place the definitions of sub-states inside the definition of a composite state. Nested definitions are only half of the story. The other half is instantiation: to enter a state, we invoke its definition. If the state is composite, when we instantiate it, it will in turn invoke (in parallel) the definitions of the sub-state it is entering. The scoping rules of π_{klt} guarantee encapsulation.

Simple transitions Events are triples $\langle p, e, d\rangle$ where $p.e \in \mathbf{Trig}$ is an event trigger and $d \in \mathbf{Vals}$ is some (optional) data associated with the event. The process for a state contains a subprocess called the *Handler*, which receives these events on the state's *inp* port. The decision of which transition to take based on this event is made by a process called *Choice*, which has a branch for each possible (sibling or outgoing) transition from the state. The result will (normally) be to execute the transition's action and then invoke the process Sn_j where n_j is the target of the corresponding transition.

Entry and exit points We represent entry points of a state via a parameter *enp* 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. The component of the state in charge of making this decision and choosing the sub-state to activate is called the *Dispatcher*.

Exit points are simpler to represent than entry points. For basic states, we do not 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 pseudostate, 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 b as Bb. Hence, if the target of a transition segment is an exit point b_j, the process will execute Bb_j, which will itself continue with the next segment in the transition chain. In addition to jumping to the destination, the Bb_j process must also inform its containing state that it is exiting, so that the containing state becomes inactive (and, more specifically, to stop the state's *Handler*).

Group transitions A group transition is a transition whose source is (an exit point of) a composite state. When taking such a transition, the currently active sub-state of the composite state becomes (recursively) inactive.

Traditionally, group transitions are usually 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 whenever one such event occurs, the *Handler* 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.

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 is a sub-state of the source of the other transition, then the conflict is resolved by giving priority to the former, inner transition (SVP 1). In this section, we implement such priority scheme. Note that this "priority" is different from the priority of events in the event pool. Such event priorities will be addressed later.

The main idea is as follows. For each composite state n, the *Handler* receives the incoming event and before it compares this event 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 as a trigger, 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) did not 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 add an *acc* and a *rej* port to inform the containing state of acceptance or rejection of events.

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 that was active when the composite state exited. This behavior is called *history*. The policy applies recursively for the sub-state, resulting in what is known as *deep history*. (SVP 2)

To implement history, we define, for each state n_k a *history cell* h_k, a process thats tores n_k's last visited sub-state. In fact, whenever we take a transition inside a composite state n_k, we store the target state of the transition n_k's history cell, and hence, h_k always contains n_k's currently active sub-state. Then, if we exit n_k and reenter it later, the *Dispatcher* recalls the state stored in the history cell.

Actions There are two main issues to be addressed in order to support actions: first, how are individual actions encoded in π_{klt}; second, when should they be executed?

To address the first question, we consider an existing set of actions **Acts** without specifying what are these actions exactly. Normally, these actions would be given in some action language (SVP 3). 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 α : **Acts** $\rightarrow$ **KLT** which maps each action to the corresponding π_{klt} term. Later on, we will provide a specific action language (Sect. 5.1.2) and a specific translation in the context of UML-RT capsules (Sect. 5.2.6).

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, exit actions must be executed whenever we exit a state. Transition actions are executed whenever the transition is taking place, after exiting the source state and before entering the target state. This means that the process *Sn* for a state $[n, \ldots, en, ex]$ must begin by executing $\alpha(en)$ and that $\alpha(ex)$ must be executed when leaving the state, in process *Bb* for each exit point *b*.

4.2.2 Formal mapping

Actions As stated above, we need a translation for actions. The particular action language may vary, so we assume that an appropriate translation is provided.

Definition 4 (*Action translation*) An *action translation* is a map $\alpha : \mathbf{Acts} \to \mathcal{C} \to \mathbf{KLT}$ from the set of possible actions **Acts** to the set of π_{klt} terms **KLT**, where $\mathcal{C}$ is some set of contextual information needed to do the translation.

History cells The history cell *h* for a given state *m* stores its last active sub-state *Sn*, as well as a boolean flag *ini*, which is set to `true` if the state *m* has been previously visited. History cells are instances of the following process:

Definition 5 (*History Cells*) History cells are represented by the following process definition:

```
proc HistoryCell(h, ini, kill, Sn) =
    when {
     h?⟨"set", Sn′⟩ → HistoryCell(h, true, kill, Sn′)
    | h?⟨"get", inp, acc, rej, exit, exack, sh, kill, enp⟩ →
        (if ini then
              Sn(inp, acc, rej, exit, exack, sh, kill, enp)
          ∥ HistoryCell(h, ini, kill, Sn))
    | h?⟨"peek", r⟩ →
        (r!ini ∥ HistoryCell(h, ini, kill, Sn))
    | kill? → done }
```

The way a history cell works is straightforward. It accepts three kinds of messages: "`set`", to store a sub-state in the cell; "`get`", to execute the currently stored sub-state; and "`peek`", to determine whether the cell has been initialized. When a "`set`" message is received, it comes with the name Sn' of the sub-state process to be stored in the cell. This is kept as the third parameter in the definition of *HistoryCell*. In this case, the *ini* flag is set to `true`, indicating that the cell has been initialized and the state has been visited at least once. When a "`get`" message arrives, if the state has been initialized, it executes the sub-state *Sn* currently stored, linking the ports and parameters passed along with the request. These parameters are explained below. Finally, when a "`peek`" message is received, it returns the value of the *ini* flag along a given channel *r*.

The process definition for states Each state n_k is translated into a process definition Sn_k, which has the following ports and parameters:

- *inp*: this is the port where input events are received,
- *acc*: this port is used to signal that an input event has been accepted by the state,
- *rej*: this port is used to signal the rejection of an input event, i.e., that the event cannot be handled by the state because no outgoing transition from this state is enabled by the event.
- *exit*: this port is used by the state's parent to request the state to exit,
- *exack*: this port is used to acknowledge an exit request, once the necessary (and possibly recursive) exit actions have been performed,
- *sh*: this port is used to signal that exiting this state also exits the enclosing state, and thus the *Handler* of the enclosing state must stop (hence *sh* for "stop handler"),
- *kill*: this port is used to stop the state and all processes associated with it, including its *Handler* and sub-states.
- *enp*: is a parameter used only in composite states to pass the name of the *entry point* used to enter the state.

The difference between *exit*, *kill*, and *sh* is as follows: *exit* is signaled when executing a group transition, so the composite state taking the transition asks its currently active sub-state to *exit* and waits for it to exit before executing the corresponding *Exit* action; *kill* is signaled when the entire state machine is being destroyed, when its capsule is being destroyed so the composite state being killed asks its active sub-state to be killed as well, without executing exit actions or waiting for sub-states to finish; and *sh* is signaled when a transition chain is being taken and going through an exit point so the handler of the composite state is to be stopped.

In the following, we assume that for each composite state n_k, there is a top-level channel h_k for its history cell. We also assume a global event *compl*, used to indicate that we have reached a stable state, and thus signaling the end of a run-to-completion step. The following definition formalizes the translation of a basic state s whose containing (parent) state is s', as $\mathcal{T}_S[\![s]\!]_{s',hist,hist',compl,ports}$, where *hist* is the link to the history cell for s, $hist'$ is the history link for its parent s', *compl* is the completion event, and $ports$ is the list of ports of the capsule containing the state machine.

Each transition is assumed to be annotated with a label (p, e, g) where p is a port name, e is an event name, and g is a guard (a boolean expression). Incoming events are of the form $\langle p, e, d\rangle$ where p is a port name, e is an event name, and d is some data associated with the event e.

This translation assumes a translation for actions α : **Acts** $\to \mathcal{C}_{ports} \to$ **KLT**, with context set $\mathcal{C}_{ports}$ whose elements are pairs $\langle\langle p, e, d\rangle, ports\rangle$ of incoming events and lists of ports (so that the action can refer or use the event and/or the capsule's ports).

We now provide the definition of the translation $\mathcal{T}_S[\![s]\!]_{...}$ for basic states in Definition 6 and composite states in Definition 7.

Translation of basic states

Definition 6 (*Translation of basic states*) Given a basic state

$s = [n_k, A, B, en, ex]$

whose parent (enclosing) state is[3]

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

and given an action translation α : **Acts** $\to \mathcal{C}_{ports} \to$ **KLT**, the translation of s is the π_{klt} term $\mathcal{T}_S[\![s]\!]_{...}$ shown in Fig. 6, where for each outgoing transition $t_i \in T''$, $\mathsf{trig}(t_i) = p_i.e_i$, and $\mathsf{guard}(t_i) = g_i$ with $T'' \overset{def}{=} \{t \in T' \mid \exists b \in B. b = \mathsf{src}(t)\}$ being the set of outgoing transitions, i.e., the transitions in the containing state s' whose source is an exit point of s, and where Q_i is the process that executes the transition's action and goes to the target of the transition, defined as

$$Q_i \overset{def}{=} \begin{cases} T_i;\ Sn_j(inp, acc, rej, exit, exack, sh, kill, a) \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{sib},\ a = \mathsf{targ}(t_i), \\ \quad \exists s_j \in S'. a \in \mathsf{entries}(s_j),\ \text{ and } n_j = \mathsf{name}(s_j) \\ T_i;\ Bb_j(sh) \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{out} \text{ and } b_j = \mathsf{targ}(t_i) \in B' \end{cases}$$

and where each T_i is the process, which executes the action of transition t_i, namely $\alpha[\![\mathsf{act}(t_i)]\!]\langle(\bot, \bot, \bot), ports\rangle$.

Explanation Figure 7 shows the control flow of the π_{klt} definition for basic states. In this definition, a basic state starts by executing its entry action *Entry*() and then it updates the history cell of its parent with the currently active state by doing $hist'!\langle$“`set`”$, Sn_k\rangle$. Hence, the history cell for s' holds the currently active sub-state s. Note that $hist'$ is the link to the parent's history cell; then, we trigger the event *compl*, signaling that we have reached a stable state, thus ending a run-to-completion step, and then start the *Handler* process, which waits for input events.

The *Handler* waits for an input event on its *inp* port, or for an *exit* request. If an event arrives on the *inp* port, the decision of what to do with it is taken by the *Choice* process, which contains a conditional with a branch for each outgoing transition with trigger $p_i.e_i$ and guard g_i. If input comes from port p_i with data event e_i and its guard g_i is true, the transition is taken: first, the *acc* event is triggered indicating that the event was accepted by the state; second, the exit action of the state (*Exit*) is executed, followed by the transition's action (T_i); and finally, control passes to the target of the transition, either some (sibling) state Sn_j (through entry point a) or some exit point (b_j) of the containing state.

[3] If the state has no parent, i.e., it is the top-most state on the state machine, the role of the parent will be taken by a special process called *Sink*, described in Definition 8.

```
$\mathcal{T}_S[\![s]\!]_{s',hist,hist',compl,ports} \overset{def}{=}$
  proc $Sn_k(inp, acc, rej, exit, exack, sh, kill, enp)$ =
    def {
      proc $Entry() = \alpha[\![en]\!]\langle\langle\bot,\bot,\bot\rangle, ports\rangle$;
      proc $Exit(p, e, d) = \alpha[\![ex]\!]\langle\langle p, e, d\rangle, ports\rangle$;
      proc $Handler()$ =
        when {
          $inp?\langle p, e, d\rangle \to\ Choice(p, e, d, acc, rej)$
          $\mid exit? \to\ Exit(\bot,\bot,\bot); exack!$
          $\mid kill? \to$ done }
      proc $Choice(p, e, d, acc, rej)$ =
        if $p$ = “p1” and $e$ = “e1” and $g_1$ then
          $acc!; Exit(p, e, d); Q_1$
        else if $p$ = “p2” and $e$ = “e2” and $g_2$ then
          $acc!; Exit(p, e, d); Q_2$
        ...
        else if $p$ = “pm” and $e$ = “em” and $g_m$ then
          $acc!; Exit(p, e, d); Q_m$
        else $rej!;\ Handler()$
    } in
      $Entry();\ hist'!\langle$“set”$, Sn_k\rangle;\ compl!;\ Handler()$
```

Fig. 6 Translation of basic states

Example 3 Consider the basic state n_3 from the state machine from Fig. 5, inside state s_2 with outgoing transitions t_3 and t_4. Figure 8 shows the result of the translation $\mathcal{T}_S[\![s_3]\!]_{s_2,hist_3,hist_2,compl,ports}$ where $ports = \langle p_1, p_2, p_3\rangle$.

In this example, one can see that the *Choice* process has three branches: one for transition t_3, one for transition t_4, and the default branch for the case when the incoming event does not match the trigger of these transitions and the event is rejected.

The first branch, corresponding to t_3, informs its containing state (Sn_2) that the event is accepted (*acc*!), executes the exit action, then executes the transition action f_1, and then executes the process corresponding to the exit point b_1 (process Bb_1). The definition for Bb_1 will be provided in the definition of the enclosing state Sn_2.

The second branch is similar, but the transition does not have an action to execute, and the target state is n_4, thus it invokes the process Sn_4 with entry point den_{n_4}, the default entry point.

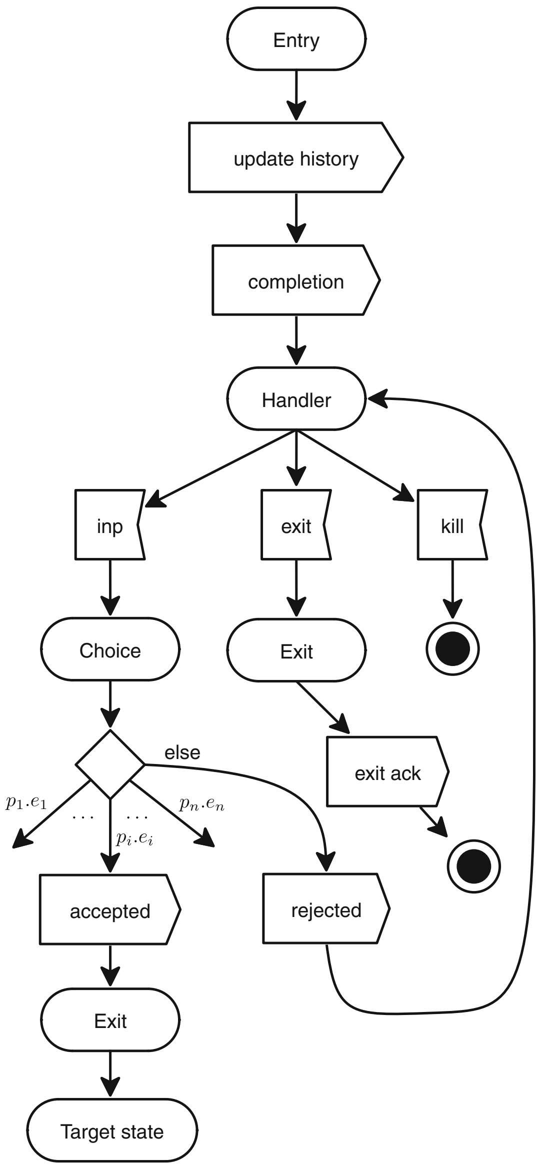

Fig. 7 Activity diagram for the control flow of basic states

In the main body, the history cell for the parent, $hist_2$, is set to this state Sn_3, after executing the entry action.

Translation of composite states

Definition 7 (*Translation of composite states*) Given a composite state

$s = [n_k, A, B, S, d, T, en, ex]$

whose parent (enclosing) state is[4]

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

[4] If the state has no parent, i.e., it is the top-most state on the state machine, the role of the parent will be taken by a special process called *Sink*, described in Definition 8.

```
$\mathcal{T}_S[\![s_3]\!]_{s_2,hist_3,hist_2,compl,\langle p_1,p_2,p_3\rangle} \overset{def}{=}$
  proc $Sn_3(inp, acc, rej, exit, exack, sh, kill, enp) =$
    def {
      proc $Entry() = \alpha[\![en]\!]\langle\langle\bot,\bot,\bot\rangle, \langle p_1,p_2,p_3\rangle\rangle$;
      proc $Exit(p, e, d) = \alpha[\![ex]\!]\langle\langle p, e, d\rangle, \langle p_1,p_2,p_3\rangle\rangle$;
      proc $Handler() =$
        when {
          $inp?\langle p, e, d\rangle \rightarrow\ Choice(p, e, d, acc, rej)$
         | $exit? \rightarrow\ Exit(\bot,\bot,\bot); exack!$
         | $kill? \rightarrow$ done }
      proc $Choice(p, e, d, acc, rej) =$
        if $p =$ "p1" and $e =$ "x" and true then
          $acc!; Exit(p, e, d)$;
          $\alpha[\![f_1]\!]\langle\langle\bot,\bot,\bot\rangle, \langle p_1,p_2,p_3\rangle\rangle; Bb_1(sh)$
        else if $p =$ "p3" and $e =$ "z" and true then
          $acc!; Exit(p, e, d)$;
          $Sn_4(inp, acc, rej, exit, exack, sh, kill,$ "den$_{n_4}$")
        else $rej!;\ Handler()$
    } in
      $Entry();\ hist_2!\langle$"set"$, Sn_3\rangle;\ compl!;\ Handler()$
```

Fig. 8 Example: translation of state n_3 from Fig. 5

```
$\mathcal{T}_S[\![s]\!]_{s',hist,hist',compl,ports} \overset{def}{=}$
  proc $Sn_k(inp, acc, rej, exit, exack, sh, kill, enp) =$
   def {
     proc $Entry() = \alpha[\![en]\!]\langle\langle\bot,\bot,\bot\rangle, ports\rangle$;
     proc $Exit(p, e, d) = \alpha[\![ex]\!]\langle\langle p, e, d\rangle, ports\rangle$;
     $D_{Handler}; D_{Forward}; D_{Choice}; D_{Dispatcher}$;
     $D_{Sn_1}; \ldots; D_{Sn_j}; D_{Bb_1}; \ldots; D_{Bb_l}$
   } in
    new $inp', acc', rej', exit', exack', sh', kill'$ in
     $(Entry()$;
      $hist'!\langle$"set"$, Sn_k\rangle$;
      $(Dispatcher(inp', acc', rej', exit', exack', sh', kill', enp)$
       $\|\ Handler(inp', acc', rej', exit', exack', sh', kill')))$
```

Fig. 9 Translation of composite states

and given an action translation $\alpha : \mathbf{Acts} \rightarrow \mathcal{C}_{ports} \rightarrow \mathbf{KLT}$, the translation of s is the π_{klt} term $\mathcal{T}_S[\![s]\!]_{\ldots}$ shown in Fig. 9, where each D_{Sn_i} is the definition of sub-state n_i:

$$D_{Sn_i} \overset{def}{=} \mathcal{T}_S[\![s_i]\!]_{s,h_i,h_k,compl,ports}$$

with h_i being the history cell for sub-state n_i (state term s_i), h_k being the history cell for n_k (state term s); D_{Bb_j} is the definition of exit point b_j given below; $D_{Handler}$, $D_{Forward}$, D_{Choice} and $D_{Dispatcher}$ are the definitions of the *Handler*, *Forward*, *Choice* and *Dispatcher* given below:

– The *Dispatcher* process definition $D_{Dispatcher}$ is given in Fig. 10, where as before, Q'_i is the target of the transition segment i, the process that executes the transition's action and goes to the target of the transition, defined as

```
proc $Dispatcher(inp', acc', rej', exit', exack',$
              $sh', kill', enp) =$
    if        $enp =$ "a1"  then  $Q'_1$
    else if   $enp =$ "a2"  then  $Q'_2$
    ...
    else if   $enp =$ "am"  then  $Q'_m$
    else      $hist!\langle$"get"$, inp', acc', rej', exit',$
                    $exack', sh', kill', enp\rangle$
```

Fig. 10 Dispatcher: chooses a sub-state according to the entry point or history

$$Q'_i \overset{def}{=} \begin{cases} T_i;\ Sn_j(inp', acc', rej', exit', exack', sh', kill', a) \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{in},\ a = \mathsf{targ}(t_i), \\ \quad \exists s_j \in S.\, a \in \mathsf{entries}(s_j), \text{ and } n_j = \mathsf{name}(s_j) \\ T_i;\ Bb_j(sh') \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{out} \text{ and } b_j = \mathsf{targ}(t_i) \in B \end{cases}$$

and where each T_i is the process that executes the action of transition t_i, $\alpha[\![\mathsf{act}(t_i)]\!]\langle(\bot, \bot, \bot), ports\rangle$.

– D_{Bb_j} is a process definition for exit point $b_j \in B$, given by

$$D_{Bb_j} \overset{def}{=} \texttt{proc}\, Bb_j(sh) = sh! \parallel Q_j$$

where sh is the parent's stop-handler signal, and Q_j is the target of the exit point, defined as follows:

$$Q_i \overset{def}{=} \begin{cases} T_i;\ Sn_j(inp, acc, rej, exit, exack, sh, kill, a) \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{sib},\ a = \mathsf{targ}(t_i), \\ \quad \exists s_j \in S'.\, a \in \mathsf{entries}(s_j), \text{ and } n_j = \mathsf{name}(s_j) \\ T_i;\ Sn_j(inp', acc', rej', exit', exack', sh', kill', a) \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{in},\ a = \mathsf{targ}(t_i), \\ \quad \exists s_j \in S.\, a \in \mathsf{entries}(s_j), \text{ and } n_j = \mathsf{name}(s_j) \\ T_i;\ Bb_j(sh) \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{out} \text{ and } b_j = \mathsf{targ}(t_i) \in B' \end{cases}$$

– $D_{Handler}$ is the process definition shown in Fig. 11.
– $D_{Forward}$ is the process definition shown in Fig. 12.
– Finally, D_{Choice} is the process definition shown in Fig. 13, where Q_i is the process that executes the transition's action and goes to the target of the transition, defined (in the same way as for Bb_j above) as

$$Q_i \overset{def}{=} \begin{cases} T_i;\ Sn_j(inp, acc, rej, exit, exack, sh, kill, a) \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{sib},\ a = \mathsf{targ}(t_i), \\ \quad \exists s_j \in S'.\, a \in \mathsf{entries}(s_j), \text{ and } n_j = \mathsf{name}(s_j) \\ T_i;\ Sn_j(inp', acc', rej', exit', exack', sh', kill', a) \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{in},\ a = \mathsf{targ}(t_i), \\ \quad \exists s_j \in S.\, a \in \mathsf{entries}(s_j), \text{ and } n_j = \mathsf{name}(s_j) \\ T_i;\ Bb_j(sh) \\ \quad \text{if } \mathsf{kind}(t_i) = \mathsf{out} \text{ and } b_j = \mathsf{targ}(t_i) \in B' \end{cases}$$

```
proc $Handler(inp', acc', rej', exit', exack',$
              $sh', kill') =$
  when {
    $inp?\langle p, e, d\rangle \rightarrow$
      new $visited$ in
        $hist!\langle$"peek"$, visited\rangle \rightarrow$
        when {
          $visited?$true $\rightarrow$
            $Forward(p, e, d, acc, rej, inp', acc', rej',$
                    $exit', exack', sh', kill')$
          | $visited?$false $\rightarrow$
            $Choice(p, e, d, acc, rej, inp', acc', rej',$
                    $exit', exack', sh', kill')$ }
  | $exit? \rightarrow exit'! \rightarrow$
      when { $exack'? \rightarrow (Exit(\bot, \bot, \bot); exack!)$ }
  | $sh'? \rightarrow$ done
  | $kill? \rightarrow kill'! \rightarrow$ done }
```

Fig. 11 Composite state handler

```
proc $Forward(p, e, d, acc, rej, inp', acc', rej',$
             $exit', exack', sh', kill') =$
  $inp'!\langle p, e, d\rangle$
  $\rightarrow$ when {
      $acc'? \rightarrow acc! \rightarrow$
        $Handler(inp', acc', rej', exit', exack',$
                $sh', kill')$
    | $rej'? \rightarrow$
        $Choice(p, e, d, acc, rej, inp', acc', rej',$
               $exit', exack', sh', kill')$ }
```

Fig. 12 Composite state event-forwarder

```
proc $Choice(p, e, d, acc, rej, inp', acc', rej',$
            $exit', exack', sh', kill') =$
  if $p =$ "p1" and $e =$ "e1" and $g_1$ then
    $exit'! \rightarrow$
      when { $exack'? \rightarrow acc! \rightarrow (Exit(p, e, d); Q_1)$ }
  else if $p =$ "p2" and $e =$ "e2" and $g_2$ then
    $exit'! \rightarrow$
      when { $exack'? \rightarrow acc! \rightarrow (Exit(p, e, d); Q_2)$ }
  ...
  else if $p =$ "pm" and $e =$ "em" and $g_m$ then
    $exit'! \rightarrow$
      when { $exack'? \rightarrow acc! \rightarrow (Exit(p, e, d); Q_m)$ }
  else $rej! \rightarrow$
    $Handler(inp', acc', rej', exit', exack', sh', kill')$
```

Fig. 13 Composite state choice taker

Explanation Figure 14 shows the control flow of the π_{klt} definition for composite states. As with basic states, the definition of a composite state contains definitions for entry and exit actions (*Entry* and *Exit*), respectively, an event handler (*Handler*) and a process to make the choice of what to do with the event (*Choice*). In addition to these, it also contains

– a *dispatcher* to either follow an incoming transition into some sub-state or recall history (*Dispatcher*),
– a definition D_{Sn_i} for each sub-state n_i,
– a definition D_{Bb_j} for each exit point b_j,

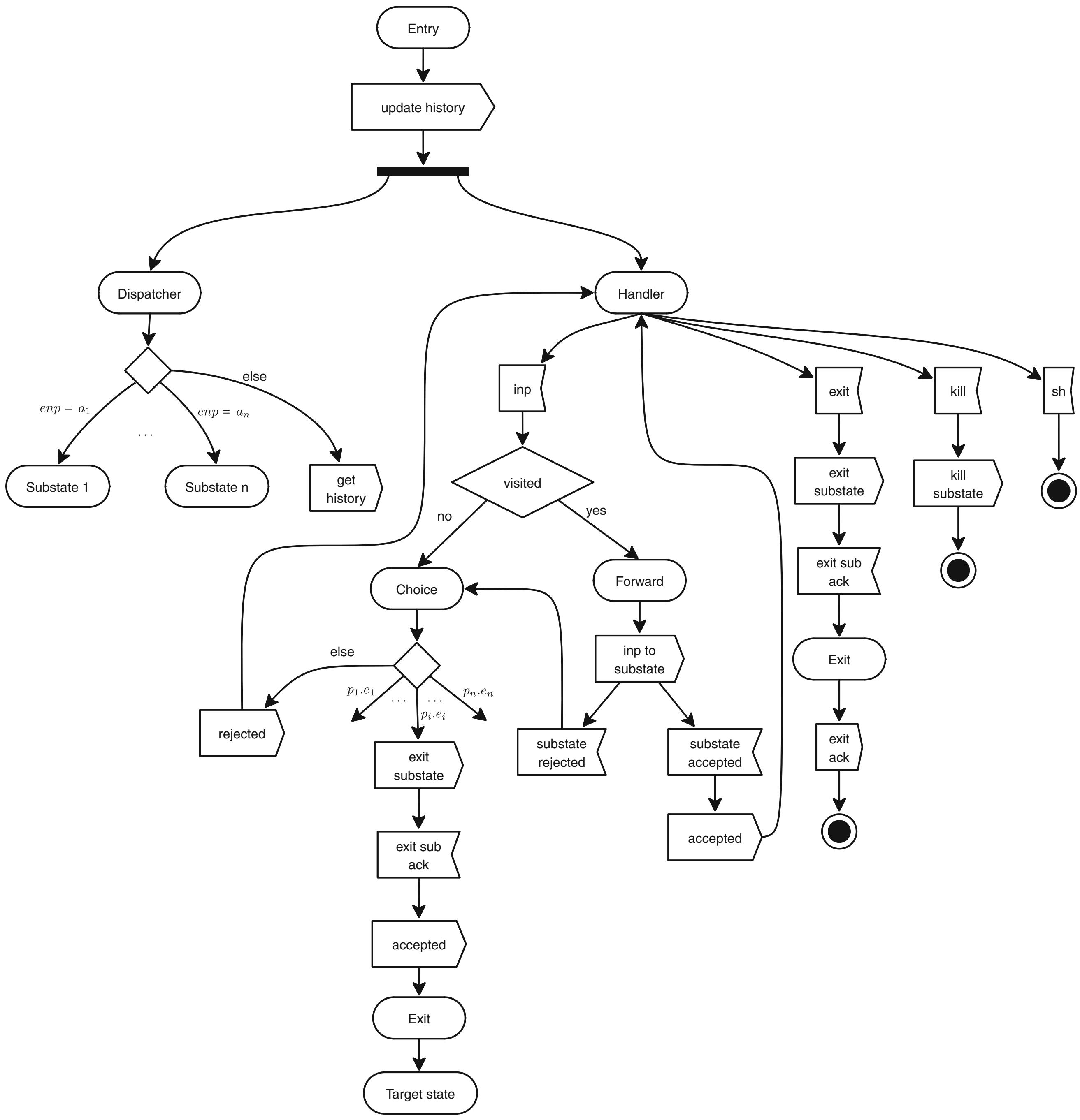

Fig. 14 Activity diagram for the control flow of composite states

– and a *forwarder* process which forwards incoming events down to the currently active sub-state, in order to implement the "deepest first" enabled-transition selection policy.

In the main body, the state begins by executing its entry action (*Entry*()) and then it updates the history cell of its parent with the currently active state by doing $hist'!\langle$"`set`", $Sn_k\rangle$, and then starts the *Dispatcher* and the event *Handler*. Both *Dispatcher* and *Handler* are invoked with primed channels inp', acc', rej', $exit'$, $exack'$, sh', and $kill'$, which are used to interact with the currently active sub-state. For example, *inp* is the channel where the composite state n_k receives its input events and inp' is the channel where n_k forwards input events to its currently active sub-state (which may change).

The *Dispatcher* chooses which sub-state to activate (invoke) based on the value of the *enp* parameter, which must contain the name of an entry point. If it is a valid entry point name, it represents an incoming transition i, thus Q'_i is performed, which executes the transition's action and invokes the process corresponding to the target of the transition, passing as parameters the current primed channels inp', acc', rej', $exit'$, $exack'$, sh', and $kill'$ so that the newly activated sub-state can interact with the *Handler*.[5] If it is not a named entry point, the history cell for the state is recalled, also passing as parameters the current primed channels inp', acc', rej', $exit'$, $exack'$, sh', and $kill'$.

The event *Handler* is somewhat more complex than that for basic states. It accepts three kinds of events: *inp* (input events), *exit*, and *sh* (stop *Handler*). On an input event, the *Handler* inquires its history cell if it has been previously visited. (SVP 4)

If the state has been previously visited, (the history cell has been initialized with some sub-state), it proceeds to the normal case handled by the *Forward* process, which forwards the event down to the currently active sub-state (via inp') and waits to see whether it was accepted (acc') or rejected (rej'). If accepted by the sub-state, the acceptance is forwarded "up" to the parent (*acc*), and we wait for the next event. If rejected by the sub-state, we attempt to handle the event at this level in the *Choice* process. (SVP 5)

If the state has not been previously visited, no sub-state has been activated, which is the case if there is no initial transition to some sub-state. In this case, the composite state remains "on the border," and thus, events are not forwarded to any sub-state and are handled by the *Choice* process.

The exit event (*exit*) might be received from the parent state, in which case the request is forwarded to the currently active sub-state via $exit'$, and when acknowledgment from the sub-state arrives ($exack'$), the exit action is executed (*Exit*) and an acknowledgment is forwarded to the parent (*exack*).

Finally, the stop *Handler* signal (*sh*) may be received when leaving the state via an exit point b_j. In this case, it is not necessary to ask the currently active sub-state to stop, because it was precisely that sub-state who has executed the outgoing transition and has performed its own exit sequence. Note that if a transition chain is executed, going through several exit points, each exit point triggers an *sh* event to stop the *Handler* of the composite state containing the exit point; thus, all *Handler*s in the chain are stopped as expected. The *kill* event is much like the stop-*Handler* event, but is used to stop the states in a "top-down" fashion: whenever the *Handler* receives a *kill* request, it kills its currently active sub-state by triggering $kill'$ and then stops. This will be used later, in the mapping for capsules, to kill a state machine when we destroy its capsule.

Example 4 Let us revisit the example from Fig. 5. Consider the composite state n_2 in that machine. The definitions for the *Handler* and *Forward* processes are fixed, only the definitions for *Dispatcher*, *Choice*, and the sub-states, as well as the exit points are specific.

First, we have the *Dispatcher*. There are three possible ways to enter the state: through entry points a_1 or a_2 or through the default entry point, in which case history will be recalled, or if the state has not been visited before, the initial state.

$$
\begin{array}{l}
\texttt{proc}\ Dispatcher(inp', acc', rej', exit', exack',\\
\qquad\qquad sh', kill', enp) =\\
\quad \texttt{if}\ enp = \text{“}\texttt{a}_1\text{"}\ \texttt{then}\\
\qquad Sn_4(inp', acc', rej', exit', exack', sh', kill', \text{“}\texttt{a}_3\text{"})\\
\quad \texttt{else}\ hist!\langle \text{“}\texttt{get}\text{"}, inp', acc', rej', exit', exack',\\
\qquad\qquad sh', kill', enp\rangle
\end{array}
$$

In the case where we enter the state through a_1, we have to take transition t_5, which has no action, and then go to state n_4, invoking Sn_4, and enter n_4 through entry point a_3. In the case where we enter through a_2, we have to recall history because there is no incoming transition connecting a_2 to any sub-state. Similarly, for the default entry point, we recall history. Note that when we recall history, if it is the first time, the resulting state will be the initial state.

Second, we have the choice process. There is only one (group) transition coming out of n_2, namely t_6. Thus, the choice process is as follows:

$$
\begin{array}{l}
\texttt{proc}\ Choice(p, e, d, acc, rej, inp', acc', rej',\\
\qquad\qquad exit', exack', sh', kill') =\\
\quad \texttt{if}\ p = \text{“}\texttt{p}_2\text{"}\ \texttt{and}\ e = \text{“}\texttt{x}\text{"}\ \texttt{and}\ \texttt{true}\ \texttt{then}\\
\qquad exit'! \to \texttt{when}\,\{exack'? \to acc! \to (Exit(p, e, d);\\
\qquad Sn_5(inp, acc, rej, exit, exack, sh, kill, \text{“}\mathsf{den}_{\mathsf{n}_5}\text{"}))\,\}\\
\quad \texttt{else}\ rej! \to Handler(inp', acc', rej',\\
\qquad\qquad\qquad exit', exack', sh', kill')
\end{array}
$$

Note that the parameters passed to Sn_5 are the unprimed ports, as this is a sibling transition, so we pass the channels used by n_2 and n_5 to communicate with their common parent (n_1), so that Sn_1 can now interact with the new currently active sub-state Sn_5.

Definitions for the exit points are also generated. The only one actually invoked by a sub-state is for b_1:

$$
\begin{array}{l}
\texttt{proc}\ Bb_1(sh) =\\
\quad sh! \parallel Sn_5(inp, acc, rej, exit, exack, sh, kill, \text{“}\mathsf{den}_{\mathsf{n}_5}\text{"})
\end{array}
$$

This is executed when taking transition t_1 (after t_3). It sends a signal *sh* to the *Handler* process of Sn_2 to stop it

[5] Note that the processes Q_i and Q'_i are very similar but differ in the parameters passed to the target state: Q'_i deals with *incoming* transitions, so the primed channels inp', acc', rej', $exit'$, $exack'$, sh', and $kill'$ are passed to the target sub-state so that it may interact with state n_k's *Handler*. On the other hand, Q_i deals with both *incoming and sibling* transitions, and therefore, in the sibling transition case, it passes on the non-primed channels *inp*, *acc*, *rej*, *exit*, *exack*, *sh*, and *kill* so that the target state becomes the currently active sub-state of its parent and thus can communicate with the parent's *Handler*.

(the parameter sh is given when exiting n_3; see the *Choice* process in Example 3, Fig. 8). Then, the process jumps to the target state Sn_5.

The rest of the definition of Sn_2 consists of the definitions of Sn_3 (Example 3, Fig. 8) and Sn_4.

The process definition of a full state machine Having defined processes for basic and composite states, we are now in a position to define the process for a whole state machine, which acts essentially as a wrapper, providing history cells, a top-level state, links to the capsule containing the state machine, a *Sink* process to catch the acceptance and acknowledgment events from the top-level state, and a process to handle kill requests, so when a kill event comes in, the states and history cells are stopped.

A full state machine is represented as a process *StateMachine* with the following ports:

- *inp*: where input events of the form $\langle p, e, d\rangle$ are received (where p is the port, e is the event, and d is some data value),
- *compl*: where the state machine signals the end of a run-to-completion step, when a stable state has been reached,
- *kill*: where requests to end the processes of the state machine are received,
- $e'_1, \ldots, e'_{n_E}, i'_1, \ldots, i'_{n_I}$: the containing capsule's (output) ports, where the actions of the state machine can send events.

The formal specification is as follows:

Definition 8 (*Translation of a full state machine*) Let $s \in$ **SM** be a state machine term. Its translation as a full state machine is $\mathcal{T}_{SM}[\![s]\!]$, given by

$$\begin{aligned}
&\texttt{proc}\ StateMachine(inp, compl, kill,\\
&\qquad\qquad e'_1, ..., e'_{n_E}, i'_1, ..., i'_{n_I}) =\\
&\quad \texttt{def}\,\{\,D_s;\ D_{HistoryCell};\ D_{Sink}\,\}\ \texttt{in}\\
&\quad\quad \texttt{new}\ acc, rej, exit, exack, sh, kill',\\
&\quad\quad\quad h_1, h_2, ..., h_{|s|}, h_{top},\\
&\quad\quad\quad kill_1, kill_2, ..., kill_{|s|}, kill_{top}\ \texttt{in}\\
&\quad\quad (HistoryCell(h_{top}, \texttt{false}, kill_{top}, \bot)\\
&\quad \parallel \textstyle\prod_{i=1}^{|s|} HistoryCell(h_i, \texttt{false}, kill_i, \bot)\\
&\quad \parallel Sn_1(inp, acc, rej, exit, exack, sh, kill', \text{“}\texttt{init}\text{"})\\
&\quad \parallel Sink(acc, rej, exit, exack, sh)\\
&\quad \parallel \texttt{when}\,\{kill? \to (kill'! \parallel kill_{top}! \parallel \textstyle\prod_{i=1}^{|s|} kill_i!)\})
\end{aligned}$$

where

- $\mathsf{name}(s) = n_1$
- $|s|$ is the number of states in s, including all sub-states,
- h_i is a link to the history cell for state n_i
- $D_{HistoryCell}$ is the definition of *HistoryCell* from Definition 5,
- D_s is the translation of s according to Definition 6 or Definition 7:

$$D_s \overset{def}{=} \mathcal{T}_S[\![s]\!]_{top,h_1,h_{top},compl,ports}$$

with $ports \overset{def}{=} \langle e'_1, \ldots, e'_{n_E}, i'_1, \ldots, i'_{n_I}\rangle$, and where top is a dummy container state term defined as $top \overset{def}{=} [n_{top}, \emptyset, \emptyset, \langle s\rangle, 1, \emptyset]$.
- and D_{Sink} is the following definition:

$$\begin{aligned}
&\texttt{proc}\ Sink(acc, rej, exit, exack, sh) =\\
&\quad \texttt{when}\,\{\\
&\quad\ acc? \to Sink(acc, rej, exit, exack, sh)\\
&\quad \mid rej? \to Sink(acc, rej, exit, exack, sh)\\
&\quad \mid exack? \to Sink(acc, rej, exit, exack, sh)\\
&\quad \mid sh? \to Sink(acc, rej, exit, exack, sh)\,\}
\end{aligned}$$

5 Capsules

We now show how capsule diagrams are encoded as kiltera processes. In this section, we begin by defining a syntax for UML-RT capsule diagrams and models (Sect. 5.1) including an action language. Then, we define the translation for this syntax (Sect. 5.2). The translation describes how to

- associate capsules to threads (Sect. 5.2.1)
- represent (thread) controllers (Sect. 5.2.2)
- represent capsules themselves (Sect. 5.2.3)
- represent ports and services (Sect. 5.2.4)
- represent optional and plug-in parts (Sect. 5.2.5)
- represent actions (Sect. 5.2.6)
- represent the timer (Sect. 5.2.7)
- put all these together (Sect. 5.2.8)

5.1 A syntax for UML-RT capsule diagrams

We use a mathematical notation for capsule diagrams, which allows us to define the mapping compositionally.

In the sequel, we will use the following sets:

- $\mathcal{N}_{cap}$: the set of all possible *capsule names*; we use $m, m_1, m_2, \ldots$ for elements in $\mathcal{N}_{cap}$;
- $\mathcal{N}_{parts}$: the set of all possible *part names*; we use $b, b_1, b_2, \ldots$ for elements in $\mathcal{N}_{parts}$
- $\mathcal{N}_{ports}$: the set of all possible *port names*; we use $p, p_1, p_2, \ldots$ for elements in $\mathcal{N}_{ports}$;
- $\mathcal{N}_{conn}$: the set of all possible *connector names*; we use $l, l_1, l_2, \ldots$ for elements in $\mathcal{N}_{conn}$;
- $\mathcal{N}_{sm}$: the set of all possible *state machine names*; we use $n, n_1, n_2, \ldots$ for elements in $\mathcal{N}_{sm}$;

- $\mathcal{N}_{lthr}$: the set of all possible *logical threads*; we use $L, L_1, L_2, \ldots$ for elements in $\mathcal{N}_{lthr}$;
- **SM**: the set of all state machine terms (defined in Sect. 4.1); $\mathbf{SM}_\bot \overset{def}{=} \mathbf{SM} \cup \{\bot\}$ is the set of state machine terms extended with the "none" value $\bot$, representing the absence of a state machine.
- **Vals**: the set of possible values (data transmitted with events between capsules).

Furthermore, we make the following assumptions about these sets:

- Every capsule is labelled with a unique name. If this is not the case, a simple traversal of the capsule diagram can give unique names, for example by providing fully qualified names or attaching a unique id.
- Within a capsule, port names and connector names are unique.

Before we define capsule diagram terms, we define the syntax for port references and connectors. We distinguish between qualified and unqualified port references. The former are used to refer to a port of a sub-capsule within the capsule of interest, while the latter is used to refer to a port of the capsule itself.

Definition 9 (*Port references and connectors*) We define the set **Portref** of *port references* according to the following BNF, with $F \in \mathbf{Portref}$:

$F ::= p$		Unqualified port reference
$\mid$	$m.p$	Qualified port reference

where $p \in \mathcal{N}_{ports}$ and $m \in \mathcal{N}_{cap} \cup \mathcal{N}_{parts}$.

We also define the set **Conn** of possible *connectors* according to the following BNF, with $k \in \mathbf{Conn}$:

$k ::= l : F \to F$ Relay or internal connector

where $l \in \mathcal{N}_{conn}$ is the name of the connector, and $F \in \mathbf{Portref}$ is a port reference. For a connector k, we define the following useful functions:

$\mathsf{name}(l : F_1 \to F_2) \overset{def}{=} l$	The name of the connector
$\mathsf{src}(l : F_1 \to F_2) \overset{def}{=} F_1$	The source of the connector
$\mathsf{targ}(l : F_1 \to F_2) \overset{def}{=} F_2$	The target of the connector

5.1.1 Capsules

Now, we can define capsule diagram terms. A capsule is fully defined by providing

- A name,
- Its ports (end, relay, and internal)
- An optional state machine
- A set of (sub-capsule) parts
- A set of connectors between ports

$c ::= [m, G, s, P, K, A]$	Capsule
$G ::= \{p_1 : w_1\, g_1, ..., p_n : w_n\, g_n\}$	Ports (or gates)
$w ::= \mathsf{w}$	Wired port
$\quad\;\; \mathsf{u}$	Unwired port
$g ::= \mathsf{end}$	External end port
$\quad\;\; \mathsf{int}$	Internal port
$\quad\;\; \mathsf{rel}$	External relay port
$P ::= \{b_1 : o_1\, m_1, ..., b_n : o_n\, m_n\}$	Sub-capsule parts
$o ::= \mathsf{fix}$	Fixed role
$\quad \mid \mathsf{opt}$	Optional role
$\quad \mid \mathsf{plug}$	Plugin role
$K ::= \{k_1, ..., k_{n'}\}$	Local connectors
$A ::= \{a_1, ..., a_{n''}\}$	Attribute names

Fig. 15 Syntax of UML-RT capsule diagrams

The following definition formalizes this by providing syntax for capsule terms.

Definition 10 (*Capsule diagram terms*) The set **CAP** of capsule diagram terms is defined according to the BNF shown in Fig. 15, where

- $m \in \mathcal{N}_{cap}$ is the name of a capsule,
- G is a set of pairs $p_i : w_i\, g_i$ where $p_i \in \mathcal{N}_{ports}$ is a port name, $w_i \in \{\mathsf{w}, \mathsf{u}\}$ and $g_i \in \{\mathsf{end}, \mathsf{int}, \mathsf{rel}\}$ is its type,
- $s \in \mathbf{SM} \cup \{\bot\}$ is a state machine term (or $\bot$ if the capsule has no state machine), (see Definition 3)
- P is the set of *sub-capsule parts* of n, more precisely a set of triples $b_i : o_i\, m_i$ where $b_i \in \mathcal{N}_{parts}$ is a part name, o_i is the part's *role* and $m_i \in \mathcal{N}_{cap}$ is a capsule name,
- $K \subseteq \mathbf{Conn}$ is a set of *connectors* subject to the conditions stated below,
- and A is a set of attribute names.

We first define the following useful functions to extract the elements of a given capsule $c = [m, G, s, P, K, A]$:

$\mathsf{name}(c) \overset{def}{=} m$	The name of the capsule
$\mathsf{ports}(c) \overset{def}{=} G$	The set of ports of the capsule
$\mathsf{capsm}(c) \overset{def}{=} s$	The capsule's state machine
$\mathsf{parts}(c) \overset{def}{=} P$	The set of sub-capsules
$\mathsf{conn}(c) \overset{def}{=} K$	The set of port connectors
$\mathsf{attrs}(c) \overset{def}{=} A$	The set of attribute names

Furthermore, we also have some functions to extract particular types of ports:

$$\begin{aligned}
\mathsf{endports}(c) &\overset{def}{=} \{p \mid p : w\,\mathsf{end} \in G\}\\
\mathsf{intports}(c) &\overset{def}{=} \{p \mid p : w\,\mathsf{int} \in G\}\\
\mathsf{relports}(c) &\overset{def}{=} \{p \mid p : w\,\mathsf{rel} \in G\}\\
\mathsf{extports}(c) &\overset{def}{=} \mathsf{endports}(c) \cup \mathsf{relports}(c)\\
\mathsf{wiredports}(c) &\overset{def}{=} \{p \mid p : \mathsf{w}\,g \in G\}\\
\mathsf{unwiredports}(c) &\overset{def}{=} \{p \mid p : \mathsf{u}\,g \in G\}
\end{aligned}$$

We generalize these functions over sets in the natural way. For example, if $C = \{c_1, c_2, \ldots, c_n\}$ is a set of capsules, then $\mathsf{name}(C) \overset{def}{=} \{\mathsf{name}(c_i) \mid c_i \in C\}$ is the set of names of all capsules in the set and it is similar for the rest of these functions.

We assume that fix is the default role, so $b : m$ is the same as $b : \mathsf{fix}\, m$. We use the notation $m.b$ to refer to the part b of capsule m, i.e., $m.b = m'$ iff $b : o\, m' \in \mathsf{parts}(c)$. We write $pcallroleb = o$ if $b : o\, m \in \mathsf{parts}(c)$.

$$\begin{aligned}
\mathsf{fixedcaps}(c) &\overset{def}{=} \{b : \mathsf{fix}\, m \in P\}\\
\mathsf{optcaps}(c) &\overset{def}{=} \{b : \mathsf{opt}\, m \in P\}\\
\mathsf{plugincaps}(c) &\overset{def}{=} \{b : \mathsf{plug}\, m \in P\}
\end{aligned}$$

We also write $l : F_1 \leftrightarrow F_2 \in K$ to mean that either $l : F_1 \rightarrow F_2 \in K$ or $l : F_2 \rightarrow F_1 \in K$. All connectors in K must satisfy the following conditions:

1. if $l : p \leftrightarrow F \in K$ where p is an unqualified port reference, then $p \in \mathsf{relports}(c) \cup \mathsf{intports}(c)$
2. if $l : m.p \leftrightarrow F \in K$ where $m.p$ is a qualified port reference, then there is a sub-capsule $c_i \in C$ such that $\mathsf{name}(c_i) = m$ and $p \in \mathsf{extports}(c_i)$
3. every port $p \in \mathsf{relports}(c) \cup \mathsf{intports}(c)$ is linked to at most one connector $k \in K$ (possibly none).
4. no port $p \in \mathsf{endports}(c)$ is linked to any connector inside c.
5. for all connectors $l : m_1.p_1 \leftrightarrow m_2.p_2$, $\{p_1, p_2\} \subseteq \mathsf{wiredports}(c)$
6. for every port $p \in \mathsf{wiredports}(c)$, there is a $l : m.p \leftrightarrow F \in K$
7. for every port $p \in \mathsf{unwiredports}(c)$, there is no $l : m.p \leftrightarrow F \in K$.

If any component of a capsule diagram is not specified, we write it as $-$. This is useful for describing partially specified models, or abstracted models.

Note that this definition does not include any reference to *protocols*. This is by design. Our goal is to provide a *behavioral semantics* of UML-RT models, but protocols play a static, syntactic role in UML-RT. Protocols can be understood as port types. As future work, we will formalize such a type system, but we leave it out of our present formalization as it would distract us from the behavioral aspects.

Definition 11 (*Capsule models*) The set **UMLRT** capsule models is defined by the following BNF:

$U ::= [c_0, c_1, \ldots, c_n]$ UML-RT model

where c_0 is designated as the model's *top capsule*, and such that for each capsule c_i, for each part $b : o\, m_j \in \mathsf{parts}(c_i)$, there is a capsule c_j such that $m_j = \mathsf{name}(c_j)$, that is, all sub-capsules must be defined in the model.

Example 5 Consider the capsule diagram from Fig. 1. Suppose that in that diagram, capsule B is fixed, capsule C is optional, and capsule D is a plug-in part. Furthermore, suppose that s_1 is the term representing the state machine of the capsule according to the syntax from Definition 3. Then, the model is represented by

$$U \overset{def}{=} [c_1, c_2, c_3, c_4]$$

where each c_i is a term representing the capsules as follows:

$$\begin{aligned}
c_1 \overset{def}{=} [&A,\ \{p_1 : \mathsf{w\,rel},\ p_2 : \mathsf{u\,end},\ p_3 : \mathsf{u\,rel},\ p_4 : \mathsf{w\,int}\}\\
&s_1,\\
&\{b_1 : \mathsf{fix}\, B,\ b_2 : \mathsf{opt}\, C,\ b_3 : \mathsf{plug}\, D\},\\
&\{l_1 : p_1 \rightarrow B.p_5,\ l_2 : p_4 \rightarrow C.p_8,\\
&\ \ l_3 : p_3 \rightarrow D.p_{11},\ l_4 : B.p_6 \rightarrow C.p_9,\\
&\ \ l_5 : B.p_7 \rightarrow D.p_{10}\}, \emptyset]\\
c_2 \overset{def}{=} [&B,\ \{p_5 : \mathsf{w}\,-,\ p_6 : \mathsf{w}\,-,\ p_7 : \mathsf{w}\,-\}, -, -, -, \emptyset]\\
c_3 \overset{def}{=} [&C,\ \{p_8 : \mathsf{w}\,-,\ p_9 : \mathsf{w}\,-,\ p_{13} : \mathsf{u}\,-\}, -, -, -, \emptyset]\\
c_4 \overset{def}{=} [&D,\ \{p_{10} : \mathsf{w}\,-,\ p_{11} : \mathsf{w}\,-,\ p_{12} : \mathsf{u}\,-\}, -, -, -, \emptyset]
\end{aligned}$$

We note the following. First, in the set of connectors, we can use either the capsule name or the name of the part (b_i) in qualified port references, as per Definition 9, so for example, $l_1 : p_1 \rightarrow B.p_5$ could have been written $l_1 : p_1 \rightarrow b_1.p_5$. Also, the direction of the arrow is not relevant, as messages can flow in either direction, so we could have written $l_1 : B.p_5 \rightarrow p_1$. We also allow a bidirectional arrow as well: $l_1 : p_1 \leftrightarrow B.p_5$; however, the designer may intend a specific direction for information flow, so the arrow can be used as a suggestion. This does not have an impact on the translation. Finally, a particular wiring of the capsules ports has been assumed.

Also note that in capsules c_2, c_3 and c_4, we have left the kinds of ports, state machines, parts, and connectors unspecified. We could interpret this as a partially specified model, or an abstracted model. However, to obtain the actual semantics with our translation, the model must be fully specified.

$$
\begin{array}{rl}
C ::= & \texttt{send}\ e(d)\ \texttt{to}\ p \\
\mid & \texttt{inform}\ p\ \texttt{in}\ t \\
\mid & \texttt{registerspp}\ p\ \texttt{on}\ s \\
\mid & \texttt{registersap}\ p\ \texttt{on}\ s \\
\mid & \texttt{deregisterspp}\ p\ \texttt{on}\ s \\
\mid & \texttt{deregistersap}\ p\ \texttt{on}\ s \\
\mid & \texttt{incarnate}\ b\ \texttt{on}\ t \\
\mid & \texttt{destroy}\ b \\
\mid & \texttt{import}\ m\ \texttt{in}\ b \\
\mid & \texttt{deport}\ m\ \texttt{from}\ b \\
\mid & \texttt{let}\ x = E\ \texttt{in}\ C \\
\mid & a := E \\
\mid & \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \\
\mid & C_1; C_2
\end{array}
$$

Fig. 16 Syntax for the action language

5.1.2 An action language

In UML-RT actions are used in state machines, but most significant actions perform operations related to capsules, such as sending messages, or creating new capsules. For this reason, we introduce a syntax for actions in this section.

The syntax presented here includes only a subset of all possible operations in UML-RT. Nevertheless, these seem to form a core subset of actions.

Definition 12 (*Actions*) The set **Acts** of all possible *actions* from Sect. 4.1 is defined according to the BNF shown in Fig. 16, (SVP 3) where C ranges over **Acts**, and where $p \in \mathcal{N}_{ports}$, $e \in \mathcal{N}_{evt}$, $d \in$ **Vals**, $b \in \mathcal{N}_{parts}$, and $t \in \mathcal{N}_{lthr}$. Expressions E can include attribute access.

Informally, these actions do the following:

- `send` $e(d)$ `to` p sends event e with data d through port p.
- `inform` p `in` t sets up a time-out event on port p after t seconds.
- `registerspp` p `on` s registers the unwired port p as an SPP with unique service name s.
- `registersap` p `on` s registers the unwired port p as an SAP with unique service name s.
- `deregisterspp` p `on` s deregisters the unwired port p as an SPP with unique service name s.
- `deregistersap` p `on` s deregisters the unwired port p as an SAP with unique service name s.
- `incarnate` b `on` t incarnates optional capsule part b on logical thread t.
- `destroy` b destroys optional capsule part b.
- `import` m `in` b imports capsule instance m in plug-in capsule role b.
- `deport` m `from` b removes capsule instance m from plug-in capsule role b.
- `let` $x = E$ `in` C declares a local variable x initialized to the value of E with scope C, and executes C.
- $a := E$ assigns the value of expression E to the capsule's instance attribute a.
- `if` E `then` C_1 `else` C_2 executes C_1 if the value of E is true, otherwise executes C_2.
- C_1; C_2 executes C_1 and then C_2.

5.2 Translating capsule diagrams

Mapping UML-RT models to π_{klt} involves the following:

- Mapping state machine diagrams to process definitions
- Mapping capsule diagrams to process definitions
- Representing UML-RT "controllers," which guide the execution of the system
- Representing the association of capsules to threads

Each of these issues is largely independent of the others, and thus, the combined map has a modular structure. We describe each of these in the following subsections.

We begin by defining the association of capsules to threads in Sect. 5.2.1 and then describe how to represent controllers in Sect. 5.2.2 followed by the encoding of capsules in Sect. 5.2.3. In Sect. 5.2.4, we detail the behavior of ports and services. In Sect. 5.2.5, we deal with optional and plug-in parts. In Sect. 5.2.6, we translate the action language into π_{klt} terms. In Sect. 5.2.7, we define the timing mechanism. Finally, in Sect. 5.2.8, we define the full translation, integrating all of the above.

5.2.1 Mapping capsules to threads

In order to support some deployment requirements, in UML-RT, it is possible to associate each capsule to a *logical thread*. Each logical thread can in turn be assigned to a *physical thread*. Each physical thread corresponds to exactly one controller, and each controller corresponds to exactly one physical thread. Hence, in addition to the UML-RT model, we must take into account

- the map from capsules to logical threads, and
- the map from logical threads to physical threads

Since the assignment to physical threads determines the controller of a capsule, this assignment is semantically meaningful, as capsules associated with different controllers will be able to execute simultaneously and use separate event pools. On the other hand, multiple logical threads on the same physical thread behave just as one logical thread. Therefore, what we are interested in is the composition of these two maps.

Let us assume that $\mathcal{N}_{lthr}$ denotes the set of possible logical thread names, and $\mathcal{N}_{pthr}$ denotes the set of physical thread names.

Definition 13 (*Capsule-to-thread assignment*) Let $N_C \subseteq \mathcal{N}_{cap}$ be a set of capsule names, $N_L \subseteq \mathcal{N}_{lthr}$ be a set of logical thread names, and $N_P \subseteq \mathcal{N}_{pthr}$ a set of physical thread names. A *capsule-to-logical thread assignment over N_C and N_L* is a function $\theta_L : N_C \rightarrow N_L$, i.e., a map from capsule names to logical thread names. A *logical-to-physical thread assignment* over N_L and N_P is a function $\theta_P : N_L \rightarrow N_P$, mapping logical thread names to physical threads. The *capsule-to-thread assignment* is the composition of these two maps: $\theta \overset{def}{=} \theta_P \circ \theta_L : N_C \rightarrow N_P$. For convenience, we also define **CAP2TH** as the set of all possible capsule-to-thread assignments.

This assignment is used in the translation by creating a new instance of a controller for each physical thread and linking a capsule m to the controller for the thread $\theta(m)$ (Sect. 5.2.8).

5.2.2 Controllers

Each capsule is associated with a *controller*. Controllers are objects, which guide the execution of a capsule or set of capsules. Controllers are responsible for implementing the run-to-completion semantics. A controller contains an event pool, and thus, all capsules associated with it share the same event pool. Hence, at any point in time, among all capsules associated with a controller, there will be only one capsule, or more precisely one state machine active, i.e., executing an event.

During execution, capsules send each other *messages* or *events*. An event is sent to a specific port in the target capsule and may have additional data associated with it, which is transmitted as part of the message.

In our mapping, capsule ports and connectors are represented as kiltera events or channels, so sending an event e with data d to a port p will be represented as triggering the event p with the pair $\langle e, d\rangle$ as parameter. On reception, if the port is a relay port, the message will go directly to the final receiver. If the port is an end port, the capsule must have a state machine. In this case, the receiving capsule forwards this information, the tuple $\langle p, e, d\rangle$ to its controller, to be queued so that it is processed when the controller decides. The capsule must also forward to the controller, the state machine's input port *smi*, and the event *smc* where the machine will signal it has completed the execution of an event. The *smc* port is called *ctrl* in the definition of states (see Definitions 6, 7 and 8). Thus, capsules send each other messages of the form $\langle e, d\rangle$, controllers queue messages of the form $\langle smi, smc, p, e, d\rangle$, and state machines expect messages of the form $\langle p, e, d\rangle$.

Definition 14 (*Controller events and event pools*) An *inter-capsule message* is a pair $\langle e, d\rangle$ where

- e is the name of a UML-RT event
- d is a reference to some data object, carried by the event

A *state machine input message* is a triple of the form $\langle p, e, d\rangle$ where

- p is the name of the target port in the receiving capsule,
- e is the name of a UML-RT event,
- d is a reference to some data object, carried by the event

A *controller message* is a tuple of the form:

$\langle smi, smc, p, e, d\rangle$

where:

- *smi* is the input port of the state machine that must deal with the event,
- *smc* is the state machine's event which signals completion,
- p is the name of the target port in the receiving capsule,
- e is the name of a UML-RT event,
- d is a reference to some data object, carried by the event

Controllers consist of two components: an *event pool* and a *Dispatcher*.

An *event pool process* is a π_{klt} process, which has the following interface:

`proc` *EventPool*(*put*, *get*)

where *put* is a port where events are received by the queue, *get* is a port to get and remove the first item in the queue. The event pool process in a controller expects, on port *put* a message of the form of controller messages described above. We do not provide a specific implementation of such process in order to leave open the particular queuing policy desired. (SVP 6)

A *Dispatcher* is a process that takes the first event available in the queue and forward it to the appropriate capsule, more specifically to the target capsule's state machine. Since the event pool holds tuples of the form $\langle smi, smc, p, e, d\rangle$ which come with the channel *smi* to the target state machine, all the *Dispatcher* has to do is to forward the tuple $\langle p, e, d\rangle$ to that channel and wait for the event *smc*, which signals that the state machine has finished processing the event. Once this *smc* event is received, the controller can process the next event in the queue. Since a new event is taken from the queue and dispatched only when the completion event has been received, the controller guarantees the run-to-completion semantics.

Definition 15 (*Controllers*) A controller is an instance of the following process:

```
proc $Controller(inp) =$
    def {
      proc $EventQueue(put, get) = Q$;
      proc $Dispatcher(qget) =$
         new $first$ in
           $(qget!first \rightarrow$
            when {$first?\langle smi, smc, p, e, d\rangle \rightarrow$
                  $smi!\langle p, e, d\rangle \rightarrow$
                  when {$smc? \rightarrow Dispatcher(qget)$}} )
    } in
      new $q$ in $(EventQueue(inp, q) \parallel Dispatcher(q))$
```

where Q is the implementation of the event pool.

5.2.3 Translating capsules

Each capsule is represented as a single process definition which, when instantiated, contains

- An instance of the capsule's state machine (called *StateMachine*)
- An instance of each sub-capsule (called Cm_i for each fixed capsule named m_i, *Opt* for each optional part and *Plugin* for each plug-in part.)
- An instance, for each port, of a port-handling process (called *WiredEIPort* and *UnwiredEIPort* depending on its type), and
- An instance of a process *CapsuleHandler*, which handles operations on the capsule itself, such as those performed by actions in the action language,

Every UML-RT connector is represented by a pair of channels: one for sending messages in each direction. For each port/channel p where input is expected, there is a port/channel p' which is used for output.

The interface of the capsule's process definitions contains a pair of ports for each end port (e_p, e'_p) and relay port (r_p, r'_p) of the capsule,[6] a *ctrl* port to link the capsule to its controller, and a *hook* channel where the capsule may receive certain instructions and queries. The *hook* channel is unique for each capsule and thus can be thought of as the capsule's address or identifier.

[6] In kiltera, channels are bidirectional, allowing both input and output on the same port. Nevertheless, we represent each UML-RT port (respectively, connector) by a pair of kiltera ports (respectively, channels) to differentiate between input and output on a port.

Internally, the definition includes the process definition corresponding to the state machine (this is called $D_{StateMachine}$ below) and a definition of the capsule handler ($D_{CapsuleHandler}$). Additionally, there is a pair of local events/channels for each internal port (i_p, i'_p) and for each port connector (l_p, l'_p). There is also a $hook_i$ channel for each sub-capsule instance (fixed, optional, or plug-in), a port handle h_p linking each port p to its port-handler process, and a local variable a_i for each attribute. Furthermore, there are local events/channels *smi*, *smc*, and *smk* representing, respectively, the state machine's input and completion, that is, *smi* is where the state machine receives events, *smc* is where the state machine signals that an event has been fully processed, *smk* used to kill the state machine when destroying the capsule.

The translation of a capsule c is parametrized by an assignment $\theta \in \mathbf{CAP2TH}$ of capsule names to controllers, or more precisely to the input channel of the capsule's controller. We also assume a global event name *sink*, used as a receptor for unconnected ports.

We will use the following conventions for naming ports and channels:

- End ports will be written as $e,\ e',\ e_1,\ e'_1,\ \ldots$
- Relay ports will be written as $r,\ r',\ r_1,\ r'_1,\ \ldots$
- Internal ports will be written as $i,\ i',\ i_1,\ i'_1,\ \ldots$
- Local connectors will be written as $l,\ l',\ l_1,\ l'_1,\ \ldots$
- Capsule attributes will be written as $a,\ a_1,\ \ldots$

Definition 16 (*Capsules to processes*) Given some UML-RT model $U = [c_0, c_1, \ldots, c_n]$ and a capsule $c = [m, G, s, P, K, A]$ in U, with $\mathsf{endports}(c) = \{e_1, \ldots, e_{n_E}\}$, $\mathsf{relports}(c) = \{r_1, \ldots, r_{n_R}\}$, $\mathsf{intports}(c) = \{i_1, \ldots, i_{n_I}\}$, parts $P = \{b_1 : o_1 m_1, \ldots, b_{n_P} : o_{n_P} m_{n_P}\}$, with each $m_i \overset{def}{=} \mathsf{name}(c_i)$ for some capsule $c_i \in U$, connectors $K = \{k_1, \ldots, k_{n_K}\}$, and attributes $A = \{a_1, \ldots, a_n\}$, we define c's translation into π_{klt} by the function $\mathcal{T}_C[\![\cdot]\!] : \mathbf{CAP} \rightarrow \mathbf{CAP2TH} \rightarrow \mathbf{KLT}$, as shown in Fig. 17, where:

- $L \overset{def}{=} \{\mathsf{name}(k_i) | k_i \in K\} = \{l_1, \ldots, l_{n_K}\}$
- $D_{StateMachine}$ is the translation of the capsule's state machine, if $s \neq \perp$, more precisely $D_{StateMachine}$ is $\mathcal{T}_{SM}[\![s]\!]$ according to Definition 8 (The e'_p and i'_p ports of the state machine are used by the state machine only to send events to sub-capsules (i'_p) or to other capsules (e'_p). All inputs to the state machine, including those from internal ports, are received through the *inp* port of the state machine, as, according to the run-to-completion semantics, a state machine must handle one and only one input event at a time.)

```
$\mathcal{T}_C[\![c]\!]\theta \overset{def}{=}$
    proc $Cm(hook, e_1, e'_1, ..., e_{n_E}, e'_{n_E},$
                 $r_1, r'_1, ..., r_{n_R}, r'_{n_R}, ctrl, async) =$
      new $smi,\ smc,\ smk,$
          $i_1, i'_1, ..., i'_{n_I},$
          $l_1, l'_1, ..., l_{n_K}, l'_{n_K},$
          $h_1, ..., h_{|G|},$
          $hook_1, ..., hook_{|P|}$ in
        def {
          $D_{StateMachine}; D_{CapsuleHandler}; D_{Body};$
          var $a_1$ = null; $\cdots$ ; var $a_{|A|}$ = null
        } in
          if $async$ then
            new $smc'$ in
              $ctrl!\langle smi, smc',$ "sys", "init", null$\rangle \rightarrow$
              when { $smi?\langle$"sys", "init", null$\rangle \rightarrow$
                       $Body(smc',$ true$)$ }
          else $Body(smc,$ false$)$
```

where D_{Body} is

```
proc $Body(smc', async) =$
     $\prod_{p_j:\mathsf{wend} \in G} WiredEIPort(h_j, p_j, p'_j, ctrl, smi, smc,$ "$\mathtt{p_j}$"$)$
  $\parallel \prod_{p_j:\mathsf{wint} \in G} WiredEIPort(h_j, p_j, p'_j, ctrl, smi, smc,$ "$\mathtt{p_j}$"$)$
  $\parallel \prod_{p_j:\mathsf{wrel} \in G} WiredRPort(h_j, p_j, p'_j, l_k, l'_k)$
  $\parallel \prod_{p_j:\mathsf{uend} \in G} UnwiredEIPort(h_j, p_j, p'_j, ctrl, smi, smc,$ "$\mathtt{p_j}$"$)$
  $\parallel \prod_{p_j:\mathsf{uint} \in G} UnwiredEIPort(h_j, p_j, p'_j, ctrl, smi, smc,$ "$\mathtt{p_j}$"$)$
  $\parallel \prod_{p_j:\mathsf{urel} \in G} UnwiredRPort(h_j, p_j, p'_j, l_k, l'_k)$
  $\parallel \prod_{b_j:\mathsf{fix}\ m_j \in P} Cm_j(hook_j, p_{j,1}, p'_{j,1}, p_{j,2}, p'_{j,2}, ..., p_{j,h}, p'_{j,h},$
                  $\theta(m_j),$ false$)$
  $\parallel \prod_{b_j:\mathsf{opt}\ m_j \in P} Opt(hook, hook_j, Cm_j,$
                  $p_{j,1}, p'_{j,1}, p_{j,2}, p'_{j,2}, ..., p_{j,h}, p'_{j,h}, ctrl)$
  $\parallel \prod_{b_j:\mathsf{plug}\ m_j \in P} Plugin(hook, hook_j, Cm_j,$
                  $p_{j,1}, p'_{j,1}, p_{j,2}, p'_{j,2}, ..., p_{j,h}, p'_{j,h})$
  $\parallel$ when { $\langle hook_1, hook_2, \cdots, hook_{|P|}\rangle? \rightarrow$
           $(CapsuleHandler(\langle hook_1, ..., hook_{|P|}\rangle)$
           $\parallel StateMachine(smi, smc, smk, e'_1, ..., e'_{n_E},$
                                    $i'_1, ..., i'_{n_I})$
           $\parallel$ when $\{smc? \rightarrow hook! \rightarrow$ if $async$ then $smc'!\})$
```

Fig. 17 Translation of capsules

– $D_{CapsuleHandler}$ is the definition of *CapsuleHandler* given in Definition 17,
– each Cm_j is the name of the process definition for the (sub)capsule named m_j, subject to the requirements described below,
– the port-handling process definitions *WiredEIPort*, *WiredRPort*, *UnwiredEIPort* and *UnwiredRPort* are given in Definition 18,
– the process definitions *Opt* and *Plugin* are given in Definitions 20 and 21 (Sect. 5.2.5). These definitions are not nested inside the capsule's definition as they are generic, independent of the capsule's specific, and so they can be defined globally, as is done in Definition 25.
– Each Cm_j is the name of the process definition for capsule c_i such that $m_j = \mathsf{name}(c_i)$. This definition is $\mathcal{T}_C[\![c_i]\!]\rho$ and is of the form:

$$\texttt{proc}\ Cm_j(hook,\ p_1,\ p'_1,\ p_2,\ p'_2,\ \ldots,\ p_h,\ p'_h,\ ctrl,\ async) = \cdots$$

In the invocation of Cm_j, the actual port sequence arguments $p_{j,1}, p'_{j,1}, p_{j,2}, p'_{j,2}, \ldots, p_{j,h}, p'_{j,h}$ is such that:
– either $p_{j,j'} \in R \cup I$ and $l : p_{j,j'} \leftrightarrow m_j.p_{j'} \in K$
– or $p_{j,j'} = l \in L$ and $l : m_j.p_{j'} \leftrightarrow m_k.p_k \in K$ for some port reference $m_k.p_k$
– or $p_{j,j'} = \mathit{sink}$ and there is no connector $l : m_j.p_{j'} \leftrightarrow F \in K$

Explanation The *Body* in the definition of a capsule creates all its parts: (1) an instance of the *CapsuleHandler* process, (2) an instance of the *StateMachine* of the capsule, (3) an instance of the *WiredEIPort* process for each wired end or internal port, (4) an instance of the *WiredRPort* process for each wired relay port, (5) an instance of the *UnwiredEIPort* process for each unwired end or internal port, (6) an instance of *UnwiredRPort* for each unwired relay port, (7) an instance of Cm_i for each fixed sub-capsule m_i, (8) an instance of *Opt* for each optional part, and (9) an instance of *Plugin* for each plug-in part. The arguments of fixed, optional, and plugin sub-capsules are such that they correspond to the connectors in the model. Figure 18 shows an overview of the structure of this process.

The *Body* of the capsule is initialized depending on the value of the parameter *async*. This boolean parameter specifies whether the capsule is in the same physical thread of its parent (i.e., connected to the same controller) or not. If *async* is false, the capsule is in the same thread, and *Body* is executed right away. If not, the capsule is initialized by sending a special "system initialization" event ("sys") to the controller, thus treating capsule initialization as any other event and passing a dummy completion event smc' to the controller. The capsule will start executing its *Body* (creating its state machine and sub-capsules) when the controller tells it that it can go ahead and do that (on reception of the message ⟨"sys", "init", null⟩).

The last part of the definition of *Body* holds the instantiation of the *CapsuleHandler* and *StateMachine* processes until all sub-capsules have triggered their *hook* event. This ensures that all sub-capsules are initialized in a bottom-up fashion. (SVP 7) Once all sub-capsules have triggered their hook, we instantiate the *StateMachine* and *CapsuleHandler* processes and we wait for the state machine to signal on its completion channel *smc* when it is ready. The *smc* event is triggered whenever the state machine reaches a stable state, namely when entering a basic state. When this event is received, the capsule can trigger its *hook* to indicate its readiness to its parent. Furthermore, if it is an asynchronous instantiation, we also trigger the dummy completion event *smc'* to tell the controller that the capsule has been created.

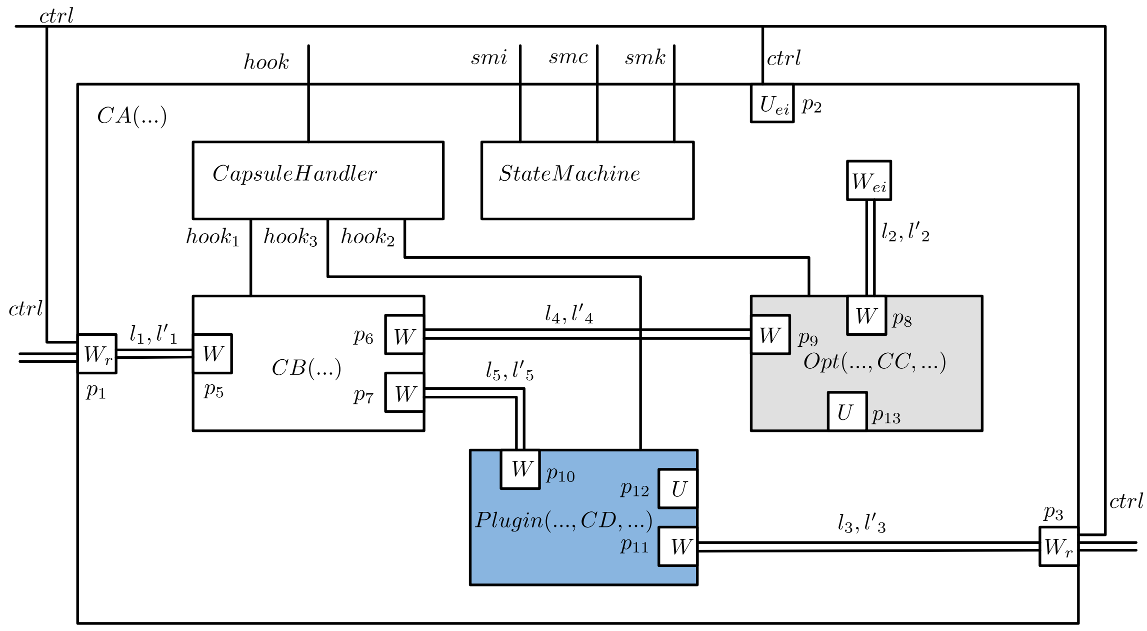

Fig. 18 Overview of the structure of the π_{klt} process of the capsule diagram in Fig. 1. Note that connectors are represented as a pair of channels (e.g., l_4, l'_4). Wired end or internal ports are labelled W_{ei}, wired relay ports are labelled W_r, unwired end or internal ports are labelled U_{ei}, and unwired relay ports are labelled U_r. For sub-capsules B, C, and D, we do not mark the subscripts ei or r, as this is specified internally in those capsules. To keep the diagram simple, we have left out several details, such as the port-handler channel h_i for each port or the $ctrl$ channel linking each sub-capsule port to their controller

Definition 17 (*Capsule handler*) The definition $D_{CapsuleHandler}$ of the capsule handler is as follows:

```
proc $CapsuleHandler(hooklist)$ =
    when {
     $hook?$“destroy" $\rightarrow$
       $(smk! \parallel \prod_{h \in hooklist} h!$“destroy"
             $\parallel \prod_{p_i : w_i g_i \in G} h_i!$“destroy")
     | $hook?\langle$“addhook", $subhook\rangle \rightarrow$
       $CapsuleHandler($list_add$(subhook, hooklist))$
     | $hook?\langle$“delhook", $subhook\rangle \rightarrow$
       $CapsuleHandler($list_del$(subhook, hooklist))$
     | $hook?\langle$“reqimport", $ports\rangle \rightarrow$
       $ports!\langle e_1, e'_1, \ldots, e_{n_E}, e'_{n_E}, r_1, r'_1, \ldots, r_{n_R}, r'_{n_R}\rangle$
        $\rightarrow CapsuleHandler(hooklist))$ }
```

The capsule *Handler* accepts requests to

- be destroyed (“destroy"), in which case it sends a kill signal to the state machine (*smk*), a “destroy" message to the hook of every sub-capsule and every port,
- add new sub-capsule hooks (“addhook") from an optional part, (see Definition 20 below)
- remove sub-capsule hooks (“delhook"),
- be imported as a plug-in elsewhere (“reqimport"), in which case it answers by providing the links to the capsules external interface.

5.2.4 Ports and services

Ports can be wired or unwired. Each port has a “handle” channel h where it receives requests to send messages, or to bind/unbind. It also has a pair of channels p^{in} and p^{out} where actual communication of messages happens. In addition, if the port is an end port or an internal port, it has a link *ctrl* to the capsule's controller, *smi* and *smc* to the capsule's state machine, and the symbolic name of the port *pname*. If it is a relay port, it a pair of links q^{in}, q^{out} to the sub-capsule connected to it.

For a wired end port or internal port, the behavior is as follows:

- When a message $\langle e, d\rangle$ arrives on the port's p^{in}, a message of the form $\langle smi, smc, pname, e, d\rangle$ is forwarded to the controller, so that the controller decides when to pass the message to the state machine. This message includes the input port of the state machine *smi* and the state machine completion event *smc* so that the controller can know where to send the actual message and wait for the event to be processed fully by the state machine.
- When a “send" request arrives on the handle channel h, the message is simply sent through the output port p^{out}.
- When an “unbind" request is received, the process switches to the unwired mode.

```
proc $WiredEIPort(h, p^{in}, p^{out}, ctrl, smi, smc, pname)$ =
  when {
   $p^{in}?\langle e, d\rangle \rightarrow$
      $ctrl!\langle smi, smc, pname, e, d\rangle$
        $\rightarrow WiredEIPort(h, p^{in}, p^{out}, ctrl, smi, smc, pname)$
  | $h?\langle$"send"$, e, d\rangle \rightarrow$
      $p^{out}!\langle e, d\rangle$
        $\rightarrow WiredEIPort(h, p^{in}, p^{out}, ctrl, smi, smc, pname)$
  | $h?$"unbind"
        $\rightarrow UnwiredEIPort(h, ctrl, smi, smc, pname)$
  | $h?\langle$"bind"$, p_1^{in}, p_1^{out}\rangle$
        $\rightarrow WiredEIPort(h, p_1^{in}, p_1^{out}, ctrl, smi, smc, pname)$
  | $h?$"destroy" $\rightarrow$ done }
```

Fig. 19 Wired end or internal ports

```
proc $WiredRPort(h, p^{in}, p^{out}, q^{in}, q^{out})$ =
  when {
   $p^{in}?\langle e, d\rangle \rightarrow$
      $q^{in}!\langle e, d\rangle$
        $\rightarrow WiredRPort(h, p^{in}, p^{out}, q^{in}, q^{out})$
  | $q^{out}?\langle e, d\rangle \rightarrow$
      $p^{out}!\langle e, d\rangle$
        $\rightarrow WiredRPort(h, p^{in}, p^{out}, q^{in}, q^{out})$
  | $h?$"unbind"
        $\rightarrow UnwiredRPort(h, q^{in}, q^{out})$
  | $h?\langle$"bind"$, p_1^{in}, p_1^{out}\rangle$
        $\rightarrow WiredRPort(h, p_1^{in}, p_1^{out}, q^{in}, q^{out})$
  | $h?$"destroy" $\rightarrow$ done }
```

Fig. 20 Wired relay ports

- When a "bind" message arrives with new channels p_1^{in}, p_1^{out}, the process simply replaces the old links with the new ones.
- When a "destroy" message arrives, the process stops.

For a wired relay port, the behavior is as follows:

- When a message $\langle e, d\rangle$ arrives on the port's p^{in}, it is resent to the sub-capsule though q^{in}.
- When a message arrives on port q^{out} from a sub-capsule, it is resent to the outside through the port p^{out}.
- Messages "bind", "unbind" and "destroy" are handled in the same way as end and internal ports.

For an unwired port, the behavior is

- When a "bind" message arrives with new channels p_1^{in}, p_1^{out}, the process switches to the wired mode with these new channels as parameters.
- When a "destroy" message arrives, the process stops.

Formally, we define these processes below.

Definition 18 (*Ports*) The definition $D_{WiredEIPort}$ for wired end or internal ports is given in Fig. 19. The definition $D_{WiredRPort}$ for wired relay ports is given in Fig. 20. The definition $D_{UnwiredEIPort}$ for unwired end or internal ports is given in Fig. 21. The definition $D_{UnwiredRPort}$ for unwired relay ports is given in Fig. 22.

```
proc $UnwiredEIPort(h, ctrl, smi, smc, pname)$ =
  when {
   $h?\langle$"bind"$, p_1^{in}, p_1^{out}\rangle$
      $\rightarrow WiredEIPort(h, p_1^{in}, p_1^{out}, ctrl, smi, smc, pname)$
  | $h?$"destroy" $\rightarrow$ done }
```

Fig. 21 Unwired end or internal ports

```
proc $UnwiredRPort(h, q^{in}, q^{out})$ =
  when {
   $h?\langle$"bind"$, p_1^{in}, p_1^{out}\rangle$
      $\rightarrow WiredRPort(h, p_1^{in}, p_1^{out}, q^{in}, q^{out})$
  | $h?$"destroy" $\rightarrow$ done }
```

Fig. 22 Unwired relay ports

The request to send a message may come directly from an action (see Definition 22), and the request to be destroyed may come from the capsule's handler (see Definition 17) but the request to bind or unbind a port always comes from a global "service handler." This is because to bind wired ports a globally unique *service name* must be provided for ports to be linked: ports can be registered as either *service provision points* (SPPs) or *service access points* (SAPs). Connections are established only between SPPs and SAPs: when an SPP is registered under the same service name as some SAP, the two become bound (connected). When an action registers an unwired port, it sends the registration request to the service handler, which keeps track of all registered ports, and links them whenever two ports match the same service name. (SVP 8)

Definition 19 (*Service handler*) The definition of the service handler process $D_{ServiceHandler}$ is as shown in Fig. 23.

The service handler receives requests in the *req* port. It keeps two dictionaries *spps* and *saps*. These dictionaries are indexed by the service name, and the values are (channels to) port handlers. When a request to register a port as an SPP arrives for a service name *sname*, the service handler looks up the service name in the *saps* dictionary. If there was an SAP already there under the service name, it creates a new pair of channels l, l' and sends a "bind" message to the port handler of both the SPP and SAP, linking them. After this, the new port handler is added to the *spps* dictionary. If there was no matching SAP, we only add the new port handler *spps*. A request to register a port as an SAP is symmetric. Deregistering is achieved in a similar fashion, sending "unbind" messages to the corresponding ports.

```
proc $ServiceHandler(req, spps, saps)$ =
 when {
  $req?\langle$"registerspp", $spp, sname\rangle \rightarrow$
   (def { var $sap$ = dict_get($sname, saps$) } in
      if $sap! =$ null then
        new $l, l'$ in
          ($sap!\langle$"bind", $l', l\rangle\rangle \parallel spp!\langle$"bind", $l, l'\rangle$)
   $\parallel ServiceHandler(req,$ dict_put($sname, spp, spps$), $saps$))
  | $req?\langle$"registersap", $sap, sname\rangle \rightarrow$
   (def { var $spp$ = dict_get($sname, spps$) } in
      if $spp \neq$ null then
        new $l, l'$ in
          ($sap!\langle$"bind", $l', l\rangle\rangle \parallel spp!\langle$"bind", $l, l'\rangle$)
   $\parallel ServiceHandler(req, spps,$ dict_put($sname, sap, saps$)))
  | $req?\langle$"deregisterspp", $spp, sname\rangle \rightarrow$
   (def { var $sap$ = dict_get($sname, saps$) } in
      ($sap!$"unbind" $\parallel spp!$"unbind")
   $\parallel ServiceHandler(req,$ dict_del($sname, spps$)
                              dict_del($sname, saps$)))
  | $req?\langle$"deregistersap", $sap, sname\rangle \rightarrow$
   (def { var $spp$ = dict_get($sname, spps$) } in
      ($sap!$"unbind" $\parallel spp!$"unbind")
   $\parallel ServiceHandler(req,$ dict_del($sname, spps$)
                              dict_del($sname, saps$))) }
```

Fig. 23 Service handler

5.2.5 Optional and plug-in parts

Definition 20 (*Optional Parts*) The definition D_{Opt} of the optional part handler process is defined as shown in Fig. 24.

In this definition, an optional part acts as a placeholder, which can receive requests to incarnate a capsule *Cm* in some thread *t* (or in the same thread as its parent, if *pthread* is *ctrl*). When a request to incarnate a new instance arrives, a message is sent to the containing capsule's handler to add the new *Cm* is instantiated with the proper connections. When a request to be destroyed arrives, the part simply stops. In this case, the parent's capsule handler takes care of destroying all created instances. This destroys only the part, not the capsules incarnated in it. Their destruction is addressed by the capsule handler itself.

Definition 21 (*Plug-in parts*) The following auxiliary process is used to connect ports:

```
proc $Plug(p_1, p_1', p_2, p_2', unplug)$ =
    when {
     $p_1'?x \rightarrow (p_2!x \parallel Plug(p_1, p_1', p_2, p_2', unplug))$
    | $p_2'?x \rightarrow (p_1!x \parallel Plug(p_1, p_1', p_2, p_2', unplug))$
    | $unplug? \rightarrow$ done }
```

The definition D_{Plugin} of the s part handler process is defined as shown in Fig. 25.

The *Plug* process simply binds two port pairs, by acting as a message forwarder, connecting the output p_1' to the input p_2 and the output p_2' to the input p_2. The *Plugin* waits for import requests which come with the imported (*targethook*).

```
proc $Opt(parenthook, hook,$
        $Cm, p_1, p_1', p_2, p_2', ..., p_h, p_h', ctrl)$ =
  when {
   $hook?\langle$"incarnate", $pthread\rangle \rightarrow$
     new $newhook$ in
      ($parenthook!\langle$"addhook", $newhook\rangle$
      $\parallel$ (if $pthread = ctrl$ then
          $Cm(newhook, p_1, p_1', p_2, p_2', ..., p_h, p_h',$
            $ctrl,$ false)
        else
          $Cm(newhook, p_1, p_1', p_2, p_2', ..., p_h, p_h',$
            $pthread,$ true))
      $\parallel Opt(parenthook, hook,$
            $Cm, p_1, p_1', p_2, p_2', ..., p_h, p_h', ctrl)$)
  | $hook?$"destroy" $\rightarrow$ done }
```

Fig. 24 Optional parts

```
proc $Plugin(parenthook, hook,$
        $Cm, p_1, p_1', p_2, p_2', ..., p_h, p_h', plugs)$ =
  when {
   $hook?\langle$"import", $targethook\rangle \rightarrow$
     new $unboundports, u_1, ..., u_h$ in
      ($targethook!\langle$"reqimport", $unboundports\rangle \rightarrow$
       when { $unboundports?\langle \bar{p}_1, \bar{p}_1', \bar{p}_2, \bar{p}_2', ..., \bar{p}_h, \bar{p}_h'\rangle \rightarrow$
           $\prod_{i=1}^{h} Plug(p_i, p_i', \bar{p}_i, \bar{p}_i', u_i)$ }
      $\parallel Plugin(parenthook, hook,$
            $Cm, p_1, p_1', p_2, p_2', ..., p_h, p_h', \langle u_1, ..., u_h\rangle$))
  | $hook?\langle$"deport", $targethook\rangle \rightarrow$
     ($\prod_{u \in plugs} u!$
     $\parallel Plugin(parenthook, hook,$
            $Cm, p_1, p_1', p_2, p_2', ..., p_h, p_h', \langle\rangle$))
  | $hook?$"destroy" $\rightarrow \prod_{u \in plugs} u!$ }
```

Fig. 25 Plugin parts

The plug-in part asks that capsule to provide a list of its (unbounded) ports to be bound. When the answer arrives, these ports are bound by the *Plug* instances. The *plugs* parameter keeps a list of the *unplug* port for each plug. When a "deport" or "destroy" message arrives, a signal is sent to unplug all plugs.

Example 6 Let us revisit the capsule from Fig. 1 and Example 5. Figure 26 shows the (top level) translation of this capsule. See also Fig. 18. Note that the order of ports in the definition is such that end ports (p_2) go first and they are followed by relay ports (p_1 and p_3). We list the connectors for internal ports (l_2, l_2') before the rest. The process *CB* is the process defined for capsule *B*, similarly for capsules *C* and *D*. Their definitions will be of the form:

```
proc $CB(hook, p_5, p_5', p_6, p_6', p_7, p_7', ctrl) = \cdots$
proc $CC(hook, p_8, p_8', p_9, p_9', p_{13}, p_{13}' ctrl) = \cdots$
proc $CD(hook, p_{10}, p_{10}', p_{11}, p_{11}', p_{12}, p_{12}', ctrl) = \cdots$
```

Note how the ports and links passed to the process invocation *CB* (in *Body*) correspond to the (positional) parameters of its definition according to the connections in the diagram.

```
$\mathcal{T}_C[\![c_1]\!]\theta \overset{def}{=}$
    proc $CA(hook, p_2, p'_2, p_1, p'_1, p_3, p'_3, ctrl) =$
      new $smi,\ smc,\ smk,$
          $l_2, l'_2,$
          $l_1, l'_1, l_3, l'_3, l_4, l'_4, l_5, l'_5,$
          $h_1, h_2, h_3, h_4,$
          $hook_1, hook_2, hook_3$ in
        def $\{D_{StateMachine}; D_{CapsuleHandler}; D_{Body}\}$ in
          if $async$ then
            new $smc'$ in
              $ctrl!\langle smi, smc', \text{"sys"}, \text{"init"}, \texttt{null}\rangle \rightarrow$
              when $\{ smi?\langle \text{"sys"}, \text{"init"}, \texttt{null}\rangle \rightarrow$
                      $Body(smc', \texttt{true}) \}$
          else $Body(smc, \texttt{false})$
```

where *Body* is

```
  proc $Body(smc', async) =$
      $WiredRPort(h_1, p_1, p'_1, l_1, l'_1)$
    $\|$ $WiredEIPort(h_4, l_2, l'_2, ctrl, smi, smc, \text{"p}_4\text{"})$
    $\|$ $UnwiredEIPort(h_2, p_2, p'_2, ctrl, smi, smc, \text{"p}_2\text{"})$
    $\|$ $UnwiredRPort(h_3, p_3, p'_3, l_3, l'_3)$
    $\|$ $CB(hook_1, l_1, l'_1, l_4, l'_4, l_5, l'_5, \theta(B), \texttt{false})$
    $\|$ $Opt(hook, hook_2, CC, l_2, l'_2, l_4, l'_4, ctrl)$
    $\|$ $Plugin(hook, hook_3, CD, l_5, l'_5, p_3, p'_3, \langle\rangle)$
    $\|$ when $\{ \langle hook_1, hook_2, hook_3\rangle? \rightarrow$
           $(CapsuleHandler(\langle hook_1, hook_2, hook_3\rangle)$
           $\|$ $StateMachine(smi, smc, smk, p'_2, l'_2)$
           $\|$ when $\{smc? \rightarrow hook! \rightarrow$ if $async$ then $smc'!\})$
```

Fig. 26 Translation of capsule c_1 from Example 5

So, for example, parameter p_5 of *CB* receives as argument p_1 since there is a relay link l_1 between them, and parameter p_6 receives l_4 the local channel that represents the connector with the same name. The same applies to the optional and plug-in parts.

The $hook_1$ links CA's capsule handler and state machine with sub-capsule B. Similarly, $hook_2$ is used to interact with part C and $hook_3$ to interact with part D.

The definition $D_{CapsuleHandler}$ of the capsule handler is as follows:

```
proc $CapsuleHandler(hooklist) =$
    when {
     $hook?$"destroy" $\rightarrow$
       $(smk! \parallel \prod_{h\in hooklist} h!$"destroy"
        $\parallel h_1!$"destroy" $\parallel h_2!$"destroy"
        $\parallel h_3!$"destroy" $\parallel h_4!$"destroy")
    | $hook?\langle$"addhook", $subhook\rangle \rightarrow$
       $CapsuleHandler($list_add$(subhook, hooklist))$
    | $hook?\langle$"delhook", $subhook\rangle \rightarrow$
       $CapsuleHandler($list_del$(subhook, hooklist))$
    | $hook?\langle$"reqimport", $ports\rangle \rightarrow$
       $ports!\langle p_2, p'_2, p_1, p'_1, p_3, p'_3\rangle$
        $\rightarrow CapsuleHandler(hooklist)$ }
```

$$
\begin{aligned}
\alpha[\![\texttt{send}\ e(d)\ \texttt{to}\ p_i]\!]c &\overset{def}{=} h_i!\langle\text{"}\texttt{send}\text{"}, e, d\rangle\\
\alpha[\![\texttt{inform}\ p\ \texttt{in}\ t]\!]c &\overset{def}{=} timer!\langle t, p\rangle\\
\alpha[\![\texttt{registerspp}\ p_i\ \texttt{on}\ s]\!]c &\overset{def}{=} shr!\langle\text{"}\texttt{registerspp}\text{"}, h_i, s\rangle\\
\alpha[\![\texttt{registersap}\ p_i\ \texttt{on}\ s]\!]c &\overset{def}{=} shr!\langle\text{"}\texttt{registersap}\text{"}, h_i, s\rangle\\
\alpha[\![\texttt{deregisterspp}\ p_i\ \texttt{on}\ s]\!]c &\overset{def}{=} shr!\langle\text{"}\texttt{deregisterspp}\text{"}, h_i, s\rangle\\
\alpha[\![\texttt{deregistersap}\ p_i\ \texttt{on}\ s]\!]c &\overset{def}{=} shr!\langle\text{"}\texttt{deregistersap}\text{"}, h_i, s\rangle\\
\alpha[\![\texttt{incarnate}\ b_j\ \texttt{on}\ L]\!]c &\overset{def}{=} hook_j!\langle\text{"}\texttt{incarnate}\text{"}, \theta_P(L)\rangle\\
\alpha[\![\texttt{destroy}\ b_j]\!]c &\overset{def}{=} hook_j!\text{"}\texttt{destroy}\text{"}\\
\alpha[\![\texttt{import}\ m_k\ \texttt{in}\ b_j]\!]c &\overset{def}{=} hook_j!\langle\text{"}\texttt{import}\text{"}, hook_k\rangle\\
\alpha[\![\texttt{deport}\ m_k\ \texttt{from}\ b_j]\!]c &\overset{def}{=} hook_j!\langle\text{"}\texttt{deport}\text{"}, hook_k\rangle\\
\alpha[\![\texttt{let}\ x = E\ \texttt{in}\ C]\!]c &\overset{def}{=} \texttt{def}\,\{\texttt{var}\,x = E\}\,\texttt{in}\,\alpha[\![C]\!]c\\
\alpha[\![x\ :=\ E]\!]c &\overset{def}{=} x\ :=\ E\\
\alpha[\![\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]c &\overset{def}{=} \texttt{if}\,E\,\texttt{then}\,\alpha[\![C_1]\!]c\,\texttt{else}\,\alpha[\![C_2]\!]c\\
\alpha[\![C_1; C_2]\!]c &\overset{def}{=} \alpha[\![C_1]\!]c; \alpha[\![C_2]\!]c
\end{aligned}
$$

Fig. 27 Mapping actions

5.2.6 Translating actions

We now present the translation α for the action language from Definition 12.

Definition 22 (*Actions to processes*) We define the map α : **Acts** $\rightarrow \mathcal{C} \rightarrow$ **KLT**, where **Acts** is the action language from Definition 12, with context set $\mathcal{C}$ whose elements are triples $\langle\langle p, e, d\rangle, ports, \theta\rangle$ of incoming events, lists of ports and thread assignment as shown in Fig. 27. In this definition:

- in the case of `inform`, the channel *timer* is the global channel to request a time-out event from the *Timer* process (Definition 23) declared in Definition 25,
- in the cases for `send` and `inform` as well as for the sap/spp (de)registering operations, h_i is the name of the handle channel for port p_i, and *shr* is the (global) request channel for the *ServiceHandler* (Definition 19), declared at the top level (Definition 25).
- in the case for `incarnate`, $hook_j$ is the name of the channel corresponding to part b_j and
- in the cases for `import` and `deport`, $hook_k$ is the m_k and $hook_j$ is the name of the channel corresponding to part b_j. Note that we assume that t is the name of a channel, which corresponds to a physical thread (see Definition 25 below).

The actions corresponding to local variables, assignment, conditionals, and sequential composition are translated directly into their corresponding constructs in kiltera. Note, however, that these constructs can themselves be expressed purely in terms of the other constructs. Here, we do not elaborate on such encoding, as it falls beyond the scope of this paper.

5.2.7 The timer

The timer process accepts requests to schedule time-out signals on a given port. When it receives a request (t, p), it will schedule an event trigger on p after a delay t. This is done asynchronously, so that multiple capsules/threads can make such scheduling requests without blocking or delaying each other.

Definition 23 (*Timer*) The definition D_{Timer} of the timer process is as follows:

$$
\begin{array}{l}
\texttt{proc}\, Timer(timer) = \\
\quad \texttt{when}\,\{\, timer?\langle time, port\rangle \rightarrow \\
\qquad\quad \texttt{wait}\; time \rightarrow port!\text{“}\texttt{timeout"} \rightarrow \\
\qquad\quad Timer(timer)\,\}
\end{array}
$$

5.2.8 The full system

The meaning and behavior of a UML-RT model depends on the assignment of capsules to threads (and therefore to controllers). Thus, the input of the translator must include

- The UML-RT model (the top-level capsule, including the definitions of all capsules and state machines): a **CAP** element.
- The maps from capsules to logical threads and to physical threads: the pair of maps θ_L and θ_P.

The kiltera process simulates the entire model by collecting all capsule definitions, and instantiating the top capsule and the controllers, with one controller for each thread.

Definition 24 (*UML-RT configuration*) A *UML-RT* configuration is a tuple $(U, N_L, N_P, \theta_L, \theta_P)$ where

- $U \in \mathbf{UMLRT}$, i.e., $U = [c_0, c_1, \ldots, c_n]$
- $c_0 \in \mathbf{CAP}$ is U's top-level capsule term,
- $N_L \subseteq \mathcal{N}_{lthr}$ is a set of logical thread names,
- $N_P \subseteq \mathcal{N}_{pthr}$ is a set of physical thread names,
- $\theta_L : N_C \rightarrow N_L$ is a capsule-to-logical thread assignment where $N_C \overset{def}{=} \{\mathsf{name}(c) \mid c \in U\}$ is the set of names of all capsules in the model,
- and $\theta_P : N_L \rightarrow N_P$ is a logical-to-physical thread assignment.

We call **UMLRTC** the set of all possible UML-RT configurations.

Now, we can provide the translation of a full input model. We create a *sink* to serve as sink for state machine events, an event *shr* where the service handler will receive requests, an event *timer* where timer will receive requests, a channel *tophook* to serve as the hook channel for the top-level capsule, and an event/channel T_i for each thread, which will be specific to each controller. The main construction simply creates an instance of the service handler, of the timer, the controllers (one for each thread), and the top-level capsule, which in turn will instantiate its sub-capsules.

Definition 25 (*Translation of a full configuration*) Given a UML-RT configuration $M = (U, N_L, N_P, \theta_L, \theta_P)$ with a model $U = [c_0, \ldots, c_k] \in \mathbf{UMLRT}$, and $N_P = \{T_1, T_2, \ldots, T_n\}$ the set of physical thread names, the translation of M is $\mathcal{M}[\![M]\!]$ where the function $\mathcal{M}[\![\cdot]\!] : \mathbf{UMLRTC} \rightarrow \mathbf{KLT}$ is defined as follows:

$$
\begin{array}{l}
\mathcal{M}[\![M]\!] \overset{def}{=} \\
\quad \texttt{new}\, sink, shr, timer\, \texttt{in} \\
\quad\; \texttt{def}\,\{ \\
\qquad D_{c_0}; D_{c_1}; D_{c_2}; \cdots ; D_{c_m}; \\
\qquad D_{Controller}; D_{ServiceHandler}; D_{Timer}; \\
\qquad D_{Opt}; D_{Plugin}; \\
\qquad D_{WiredEIPort}; D_{UnwiredEIPort}; \\
\qquad D_{WiredRPort}; D_{UnwiredRPort}; \\
\quad\; \}\,\texttt{in} \\
\qquad \texttt{new}\, tophook, T_1, T_2, ..., T_n\, \texttt{in} \\
\qquad\; (Cm_0(tophook, \theta(m_0)) \\
\qquad\quad \parallel ServiceHandler(shr, \texttt{empty\_dict}, \\
\qquad\qquad\qquad\qquad\qquad \texttt{empty\_dict}) \\
\qquad\quad \parallel Timer(timer) \\
\qquad\quad \parallel \prod_{i=1}^{n} Controller(T_i))
\end{array}
$$

where $m_0 = \mathsf{name}(c_0)$ is the top capsule's name, $\theta \overset{def}{=} \theta_P \circ \theta_L$ is the capsule-to-thread assignment,

$$D_{c_i} \overset{def}{=} \mathcal{T}_C[\![c_i]\!]\theta$$

is the translation of capsule c_i (see Definition 16) and where $D_{Controller}$ is the definition of *Controller* (see Definition 15), $D_{ServiceHandler}$ is the definition of the *ServiceHandler* process from Definition 19, D_{Timer} is the definition of *Timer* from Definition 23, D_{Opt} is the definition of *Opt* (see Definition 20), D_{Plugin} is the definition of *Plugin* (see Definition 21), and $D_{WiredEIPort}$ and $D_{UnwiredEIPort}$ are the definitions of the processes for port-handling given in Definition 18.

Example 7 Let us finish by revisiting Example 1. Assume that c_0, c_1, c_2 are the representations of capsules A, B and C, and L_i and T_i are the names of logical and physical threads. The model, or more precisely, the configuration with all capsules mapped to the same physical thread is

$$
\begin{array}{l}
M_0 = (\{c_0, c_1, c_2\}, \{L_0, L_1\}, \{T_0, T_1\}, \\
\quad \{A \mapsto L_0, B \mapsto L_0, C \mapsto L_1\}, \{L_0 \mapsto T_0, L_1 \mapsto T_0\})
\end{array}
$$

and the configuration with capsule C mapped to a different physical thread is

$$
\begin{array}{l}
M_1 = (\{c_0, c_1, c_2\}, \{L_0, L_1\}, \{T_0, T_1\}, \\
\quad \{A \mapsto L_0, B \mapsto L_0, C \mapsto L_1\}, \{L_0 \mapsto T_0, L_1 \mapsto T_1\})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{M}[\![M_i]\!] \overset{def}{=} \\
\quad \texttt{new}\ sink,\ shr,\ timer\ \texttt{in} \\
\quad\quad \texttt{def}\ \{ \\
\quad\quad\quad D_{c_0}; D_{c_1}; D_{c_2}; \\
\quad\quad\quad D_{Controller}; D_{ServiceHandler}; D_{Timer} \\
\quad\quad\quad D_{Opt}; D_{Plugin}; \\
\quad\quad\quad D_{WiredEIPort}; D_{UnwiredEIPort}; \\
\quad\quad\quad D_{WiredRPort}; D_{UnwiredRPort}; \\
\quad\quad \}\ \texttt{in} \\
\quad\quad\quad \texttt{new}\ tophook, T_0, T_1\ \texttt{in} \\
\quad\quad\quad\quad (CA(tophook,\ T_0) \\
\quad\quad\quad\quad\ \| \ ServiceHandler(shr, \texttt{empty\_dict}, \texttt{empty\_dict}) \\
\quad\quad\quad\quad\ \| \ Timer(timer) \\
\quad\quad\quad\quad\ \| \ Controller(T_0) \ \| \ Controller(T_1))
\end{array}
$$

Fig. 28 Generated π_{klt} model for M_i (Example 1 with two physical threads)

In both cases, C is associated with logical thread L_1 but in the first, L_1 is assigned to physical thread T_0 (the same as A and B), whereas in the second it is assigned to T_1. Then, the resulting π_{klt} according to Definition 25 is shown in Fig. 28. So while in both cases we have two physical threads (and two controllers), the second controller is used only in M_1. More precisely, the incarnate action in state n_2 of A (`incarnate` C `on` L_1) is translated according to Definition 22 into the term $hook_2!\langle$"`incarnate`", $\theta_P(L_1)\rangle$ where $hook_2$ is the input channel for CC's control handler. This means that for M_0, this action is $hook_2!\langle$"`incarnate`", $T_0\rangle$ and for M_1, it is $hook_2!\langle$"`incarnate`", $T_1\rangle$. Hence, according to Definition 20, when the *Opt* process inside the top-level capsule *CA* receives this incarnation message, it will create a *newhook* link to the new capsule instance and send an $\langle$"`addhook`", *newhook*$\rangle$ message to *CA*'s capsule handler. Then, it will invoke the process *CC* to instantiate the new capsule, but in the first case the invocation will be $CC(newhook, p_4, p'_4, T_0, \texttt{false})$, whereas in the second case, it will be $CC(newhook, p_4, p'_4, T_1, \texttt{true})$. As a result, in the first case, the instance of *CC* will communicate with the controller on T_0, thus sharing the same event pool with *CA* and *CB*, and in the second, with the controller on T_1 with its own separate event pool. We could modify the example to have only one physical thread and one or more logical threads with the same effects.

6 Related work

There have been many approaches proposed in the literature, aiming to formalize different aspects of UML. For example [13], proposes a semantics of activity diagrams using labelled transition systems, while [59] presents a semantics of activity diagrams using Petri Nets. In [32], a semantics of sequence diagrams is proposed in terms of certain kind of transition system, while [8] uses a custom temporal logic for defining the semantics of sequence diagrams. Aspects of UML state machines have been formalized, among others, in [33] using Generalized Stochastic Petri Nets, in [45] using term rewriting systems, in [39] and in [62] using CSP, or in [37] using LOTOS. A semantics for a kernel action language for UML has been proposed in [14] using labelled transition systems.

Other related work includes [9] where a subset of the UML for real-time systems called krtUML is proposed and its semantics formalized with symbolic transition systems. In [58], a semantics is presented for a subset of UML consisting of flat state machines and sequence diagrams with no hierarchical structure diagrams using linear temporal logic. Möller et al. [36] studies a real-time extension of UML state machines providing a semantics in terms of timed-automata. In [65], a semantics for a real-time variant of standard UML state machines is presented in terms of transition systems.

The book UML 2 Semantics and Applications [30] includes several articles proposing formal semantics for fragments of UML 2, including non-flattening semantics for state machines. Nevertheless, UML-RT itself is not the same as UML or real-time UML, and work formalizing it is less common.

A number of papers have presented formal semantics for small subsets of UML-RT using either CSP or some timed variant of CSP [1,7,12,15,16].

In [16], only capsule diagrams are translated into CSP processes, assuming synchronous communication (the default in CSP), no state machines, and no support for dynamic features such as optional or plug-in capsules, dynamic wiring or thread assignments.

In [12], a translation to CSP is also provided with the aim of studying the preservation of consistency in model evolution. This translation deals only with flat state machines and flat structure diagrams and no dynamic structure features or thread assignments.

The translation in [15] actually goes in the opposite direction, from CSP processes to UML-RT.

In [7], and later in [1], a transformation of UML-RT models into a timed variant of CSP called CSP+T is proposed. It addresses hierarchical state machines but without group transitions or history, and like the previous papers, it relies on CSP's synchronous communication and has no support for dynamic features or thread assignment.

A similar approach has been proposed in [52] where the target language is Circus, a combination of CSP and Z. This translation suffers from the same limitations of the previous ones.

It should be noted that all existing approaches to the semantics of UML-RT state machines, unlike ours, flatten the state machine. This forgets the hierarchical structure, which in turn means that there is no obvious way to encode the priority of inner transitions over outer transitions. Furthermore,

Table 1 Comparison of formal semantics for UML-RT

	Features	[61]	[31]	[4,5]	Ours
Underlying semantics		LTS	AsmL	π-calculus	π_{klt}-calculus
Semantics definition		SOS	Hard coded	By example	By translation
Executable		No	Yes	No	Yes
State machines	Hierarchy	Yes	No	No	Yes
	Group transitions	Yes	No	No	Yes
	Deep history	Yes	No	No	Yes
	Enabled-transition selection	Yes	No	No	Yes
Action language		No	Subset	Subset	Subset
Timer service		No	No	No	Yes
Capsules	Fixed capsules	Yes	Yes	Yes	Yes
	Optional capsules	No	Yes	No	Yes
	Plugin capsules	No	Yes	No	Yes
	Attributes	No	Yes	No	Yes
	Services	No	No	No	Yes
	Dynamic wiring	No	Yes	No	Yes
	Capsule-thread assignment	No	Yes	No	Yes
	Multiple controllers	No	Yes	No	Yes

it complicates traceability between model elements and the generated artifact.

A very different approach is presented in [18] where the semantics of a very small subset of UML-RT is described as an algebra of flow graphs. This is an interesting approach, but not only is it limited in its scope and coverage of UML-RT but it is also unclear how it could be leveraged for the analysis.

The work most closely related to our own is that of [31,61] and [4,5]. Table 1 summarizes the main differences between these and our semantics.

In [61], a formal semantics for a subset of UML-RT is presented using Structural Operational Semantics (SOS) to define a labelled transition system (LTS) as the semantic domain. This has the advantage that the meta-theory for SOS over LTSs is well developed. On the other hand, it does not deal with many essential aspects of UML-RT such as optional and plug-in capsules, dynamic wiring or capsule-to-thread assignments, and no action language is given. Furthermore, that paper also distinguishes between basic capsules (without sub-capsules), non-behavioral capsules (without a state machine), and behavioral capsules, whereas we do not make such a distinction and the three cases are treated uniformly.

In [4,5], a semantics for UML-RT is proposed using the π-calculus. However, this considers only a very small subset of UML-RT, without hierarchical state machines, no group transitions or history, a very limited form of rewiring, no optional or plug-in capsules, no threads or controllers, and no attributes. Furthermore, the presentation of this semantic mapping is by example only without an actual formal mapping or other systematic way of translating models into π-calculus terms.

The work in [31] is much more elaborate with respect to UML-RT than any other attempt. The authors propose a semantics for UML-RT in terms of AsmL, an object-oriented language based on Abstract State Machines (ASMs). They propose an architecture to support alternative semantics for UML-RT by relying on object-oriented polymorphism in AsmL. Different UML-RT concepts are represented as AsmL classes. Unlike all previous papers, they support optional and plug-in classes as well as multiple controllers. However, they only support flat state machines and no dynamic wiring of SPPs and SAPs. Furthermore, they do not provide an automatic translation of UML-RT models, so the modeller must manually represent the model as an AsmL data structure.

Finally, we cite our previous work [10] where we introduced early version of the mapping of state machines without history or enabled-transition selection policy, and without support for capsules.

7 Concluding remarks

We have proposed a formal syntax and semantics for the UML-RT language in terms of a process algebra called kiltera. We believe this is the most comprehensive formalization of the semantics of UML-RT to date. Unlike existing attempts, our formalization deals with both fully hierarchical state machines and structure diagrams. On both aspects, it supports features not available in other approaches, such as history or group transitions in state machines, or optional capsules and dynamic wiring in capsule diagrams. It is the only semantics with explicit support for thread assignment.

In addition to these contributions, it should also be noted that unlike much of the existing approaches, we provide an actual mapping specifying the translation, whereas some papers simply propose their semantics in an ad hoc by-example manner, without providing an actual translation. Furthermore, our mapping has been implemented using IBM RSA's transformation tool, providing a realization of the semantics. The outcome of this translation can be used by the implementation of kiltera for simulation. The development of this implementation itself helped validating the translation. We are currently working toward a kiltera model-checker, which will provide analysis capabilities. The modular nature of the translation, including mapping of states and capsules to processes, can be leveraged by the model-checker for the purpose of traceability, by providing a simple way to link the results of analysis on the generated kiltera code to the corresponding model elements.

Semantic variation points are an issue whenever we attempt to formalize a language for which the semantics has been only partially given. This is the case with the UML in general and with UML-RT in particular. We have attempted to define our mapping as precise as possible while marking semantic variation points explicitly. Nevertheless, it is important to keep in mind that there are different kinds of variation points, some of which can be easily addressed and some which would require major changes in the mapping. For example, the action language can be changed or extended with relative ease simply by providing an alternative mapping α to be invoked by the translation of state machines in Definitions 6 and 7. The translation of state machines themselves can be replaced in its entirety by providing an alternative definition of $\mathcal{T}_{SM}$ (Definition 8), which is invoked by the translation of capsules in Definition 16. However, some semantic variation points require more delicate "surgery." For example, changing the enabled-transition selection policy from inside-out to outside-in would require replacing the process *Handler* in the definition for composite states Definition 7. Changing it to full non-determinism would imply an even more radical change, even eliminating the need for the accept/reject protocol. Similarly, in the mapping of capsules (Definition 16), details of the difference between incarnation in the same thread or in a different thread, or the bottom-up initialization of capsules could be changed by an alternative definition of $\mathcal{T}_C$.

One aspect that we did not touch was the encoding of protocols. This is because our mapping encodes the *dynamic behavior* of UML-RT, while protocols (in UML-RT) contain only *static* type information, specifically the type of events allowed in a given port. In other words, our mapping assumes that the input model is well-typed, and under such assumption, it will give the model's behavior.

We have proposed a semantics of UML-RT by means of translation to another language with a well-defined formal semantics. In the introduction, we motivated our choice of kiltera as the target language on the basis of its conceptual similarities to UML-RT. Nevertheless, in spite of these similarities, the mapping is not trivial. This observation is important because most of the existing work on formalizing rich, expressive, and realistic languages tends to oversimplify them, overlooking many aspects that are often deemed "irrelevant" or "implementation-specific" but turn out to be semantically meaningful, as is the case with thread assignments in UML-RT. A common mistake is to assume that similar concepts are mapped in a one-to-one fashion between the source and the target language, but the thread assignment issue illustrates the problem. For example, in [4,5], capsules are mapped onto π-calculus processes without regard to their thread assignment. This would be fine but only under the assumption that each capsule executes on a separate physical thread. Otherwise, analysis of the resulting π-calculus processes would fail to detect possibilities for deadlock, or behaviors depending on message ordering. In short, it would lead to incorrect analysis results. This highlights the perils of using minimalistic and pure languages or formalisms to define semantics of realistic languages.

Artifacts Our kiltera simulator prototype is available at http://www.kiltera.org. Some sample UML-RT models (for IBM's RSA-RTE) together with their generated kiltera code and their meta-models are available at the Repository for Model-Driven Development (ReMoDD: http://www.cs.colostate.edu/remodd/v1/). The actual transformation is available directly from the authors.

Acknowledgments We are indebted to Bran Selic for his support and his insights into the semantics of UML-RT. We also thank Malina Software, IBM Canada, and the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Centres for Excellence (OCE), which provided financial support. We would also like to thank Eyrak Paen for his implementation of the transformation, which was instrumental in the proper definition of the mapping. Finally, Karolina Zurowska and Eric Rapos also provided very useful input on RoseRT and RSA-RTE.

Appendix: Semantic variation points

1. Alternative semantics could include giving priority to the states higher in the hierarchy, or to leave the choice as non-deterministic.
2. In UML 2, alternative semantics include (1) *shallow history*, remembering only the immediate sub-state; (2) allowing both deep and shallow history; (3) no history.

3. The action language is a major semantic variation point, but it should include at least an action to send messages. Other common actions concern operations on capsules such as accessing/modifying attributes, incarnating/destroying optional sub-capsules, or rewiring ports. IBM RSA-RTE supports three action languages: C++, Java, and UAL (UML Action Language), a Java-like language closely related to the OMG ALF standard.
4. This may be treated in a different way and handle the event in the same way regardless of whether the state was previously visited.
5. The forwarding of events down to the active sub-state is done in order to account for the priority of inner enabled transitions over outer transitions. A different priority scheme would be changed here. For example, giving outer transitions priority would attempt the *Choice* process first and if no alternative was there, the *Forward* process would be tried instead. Allowing non-deterministic choice between transitions at different levels of nesting would require a different approach with no forwarding involved.
6. There are many possible implementations of the event pool, of which the most natural would be a priority queue, where the priority is an attribute of the event itself.
7. Alternatively, this could be changed to initializing the top first and then the sub-capsules, or a more general approach allowing initialization in any order.
8. In this definition, we allow only binary connection, i.e., each connector links only two ports, and ports have multiplicity 1. To support n-ary multiplicity, the definition of the service handler should be adapted accordingly.

References

1. Benghazi Akhlaki, K., Capel Tuñón, M.I., Holgado Terriza, J.A., Mendoza Morales, L.E.: A methodological approach to the formal specification of real-time systems by transformation of UML-RT design models. Sci. Comput. Program. **65**, 41–56 (2007)
2. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Inf. Control **60**(1–3), 109–137 (1984)
3. Bertolino, A., De Angelis, G., Bartolini, C., Lipari, G.: A UML Profile and a Methodology for Real-Time Systems Design. Technical Report, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" (2005)
4. de Melo Bezerra, J., Hirata, C.M.: A Semantics for UML-RT Using π-Calculus. In: Proceedings of International Workshop on Rapid System Prototyping (RSP'07) (2007)
5. de Melo Bezerra, J., Hirata, C.M.: A polyadic pi-calculus approach for the formal specification of UML-RT. Adv. Softw. Eng. **2009**, (2009). doi:10.1155/2009/656810
6. Boudol, G.: Asynchrony and the π-Calculus (Note). Technical Report 1702, INRIA-Sophia Antipolis (1992)
7. Capel, M.I., Mendoza, L.E., Akhlaki, K.B., Holgado, J.A.: A semantic formalization of UML-RT models with CSP+T processes applicable to real-time systems verification. In: Proceedings of Jornadas de Ingeniería del Software y Bases de Datos (JISBD'06), pp. 283–292 (2006)
8. Cho, S.M., Kim, H.-H., Cha, S.D., Bae, D.-H.: A semantics of sequence diagrams. Inf. Process. Lett. **84**(3), 125–130 (2002)
9. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: Understanding UML: a formal semantics of concurrency and communication in real-time UML. In: Proceedings of FMCO'02. LNCS, pp. 71–98. Springer, Berlin (2002)
10. Dingel, J., Paen, E., Posse, E., Rahman, R., Zurowska, K.: Definition and implementation of a semantic mapping for UML-RT using a timed pi-calculus. In: Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications, BM-FA '10, pp. 1:1–1:8. ACM, New York, NY (2010)
11. Douglass, B.P.: Real-time UML. Formal Techniques in Real-Time and Fault-Tolerant Systems, Volume 2469 of LNCS, pp. 53–70. Springer, Berlin (2002)
12. Engels, G., Heckel, R., Küster, J.M., Groenewegen, L.: Consistency-preserving model evolution through transformations. In: Proceedings of the Fifth International Conference on the Unified Modeling Language—The Language and its Applications, pp. 212–227. Springer, Berlin (2002)
13. Eshuis, R., Wieringa, R.: A Formal Semantics for UML Activity Diagrams—Formalising Workflow Models. Technical Report. University of Twente (2001)
14. Fecher, H., Kyas, M., De Roever, W.-P., De Boer, F.S.: Compositional operational semantics of a UML-Kernel-model language. Electron. Notes Theor. Comput. Sci. **156**, 79–96 (2006)
15. Ferreira, P., Sampaio, A., Mota, A.: Viewing CSP specifications with UML-RT diagrams. Electron. Notes Theor. Comput. Sci. **195**(0), 57–74 (2008). Proceedings of the Brazilian Symposium on Formal Methods (SBMF 2006)
16. Fischer, C., Olderog, E.-R., Wehrheim, H.: A CSP view on UML-RT structure diagrams. In: Proceedings Fundamental Approaches to Software Engineering (FASE'01), Volume 2029 of LNCS, pp. 91–108. Springer, Berlin (2001)
17. Garlan, D., Monroe, R.T., Wile, D.: Acme: architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, Chapter 3, pp. 47–67. Cambridge University Press, New York, NY (2000)
18. Grosu, R., Broy, M., Selic, B., Stefanescu, G.: Behavioral Specifications of Businesses and Systems, Chapter 6: What is Behind UML-RT?, pp. 73–88. Kluwer, Dordrecht (1999)
19. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
20. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: Proceedings of ECOOP '91, Volume 512 of LNCS, pp. 133–147. Springer, Berlin (1991)
21. IBM: General Description Language. IBM, 9 March (2005)
22. IBM: IBM Rational Rose Technical Developer, Version 7.0. IBM, (2010) http://www-01.ibm.com/software/awdtools/developer/technical
23. IBM: IBM Rational Software Architect, RealTime Edition, Version 7.5.2. IBM (2010) http://publib.boulder.ibm.com/infocenter/rsarthlp/v7r5m1/index.jsp
24. IEEE Computer Society: IEEE Standard Verilog® Hardware Description Language, IEEE Standard 1364$^{\text{TM}}$-2001, 28 September (2001)
25. IEEE Computer Society: IEEE Standard VHDL Language Reference Manual, IEEE Standard 1076$^{\text{TM}}$-2008, 26 January (2009)
26. IEEE Computer Society: IEEE Standard for the SystemC Language, IEEE Standard 1666$^{\text{TM}}$-2011, January (2012)
27. IEEE Computer Society: IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Standard 1800$^{\text{TM}}$-2012, 21 February (2013)
28. International Telecommunications Union: Specification and description language (SDL). ITU-T Recommendation Z.100., November (1999)
29. Jefferson, D.R.: Virtual time. ACM-TOPLAS **7**(3), 404–425 (1985)

30. Lano, K., Clark, D.: UML 2 Semantics and Applications. Chapter Ch. 8—Axiomatic Semantics of State Machines, pp. 179–204. Wiley, New York (2009)
31. Leue, S., Stefanescu, A., Wei, W.: An AsmL semantics for dynamic structures and run time schedulability in UML-RT. In: Paige, R.F., Meyer, B. (eds.) Proceedings of Objects, Components, Models and Patterns (TOOLS EUROPE 2008), Volume 11 of Lecture Notes in Business Information Processing, pp. 238–257. Springer, Berlin (2008)
32. Li, X., Liu, Z., Jifeng, H.: A formal semantics of UML sequence diagrams. In: Proceedings of the 2004 Australian Software Engineering Conference, pp. 168–177 (2004)
33. Merseguer, J., Bernardi, S., Campos, J., Donatelli, S.: A compositional semantics for UML state machines aimed at performance evaluation. In: Proceedings of the 6th International Workshop on Discrete Event Systems, pp. 295–302. IEEE Computer Society Press (2002)
34. Milner, R.: A Calculus of Communicating Systems. Springer, Berlin (1980)
35. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. Reports ECS-LFCS-89-85 and ECS-LFCS-89-86 86. Computer Science Dept., University of Edinburgh (1989)
36. Möller, M.O., David, A., Yi, W.: Verification of UML statechart with real-time extensions. In: Fundamental Approaches to Software Engineering (FASE'2002), Volume 2306 of LNCS, pp. 218–232. Springer, Berlin (2003)
37. Mrowka, R., Szmuc, T.: UML Statecharts Compositional Semantics in LOTOS. In: 2008 International Symposium on Parallel and Distributed Computing, pp. 459–463. IEEE Computer Society Press (2008)
38. Muthiayen, D.: Real-Time Reactive System Development: A Formal Approach Based on UML and PVS. PhD thesis. Concordia University (2000)
39. Ng, M.Y., Butler, M.: Towards formalizing UML state diagrams in CSP. In: Proceedings of SEFM'03, pp. 138–147. IEEE Computer Society (2003)
40. Object Management Group: UML Profile For Schedulability, Performance, And Time v1.1. http://www.omg.org/spec/SPTP/, January (2005)
41. Object Management Group: UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems v1.1. http://www.omg.org/spec/MARTE/, June (2011)
42. Object Management Group: UML Superstructure Specification v2.4.1. http://www.omg.org/spec/UML/2.4.1/, August (2011)
43. Object Management Group: OMG Systems Modeling Language (OMG SysML^{TM}). http://www.omg.org/spec/SysML/1.3/ June (2012)
44. Object Management Group: UML Superstructure Specification v2.5. http://www.omg.org/spec/UML/2.5/ September (2012)
45. Paltor, I.: The Semantics of UML State Machines. Technical report (1999)
46. Posse, E.: Modelling and Simulation of Dynamic Structure, Discrete-Event Systems. Ph.D. Thesis. School of Computer Science, McGill University (2008)
47. Posse, E.: A Real-Time Extension to the π-calculus. Technical Report 2009-557. School of Computing, Queen's University, http://www.cs.queensu.ca (2009)
48. Posse, E.: The π_{klt}-Calculus: Formal Definition. Technical Report 2012-591, School of Computing, Queen's University, http://www.cs.queensu.ca, July (2012)
49. Posse, E., Dingel, J.: `kiltera`: a language for timed, event-driven, mobile and distributed simulation. In: Proceedings of the 14th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2010) (2010)
50. Posse, E., Dingel, J.: Theory and implementation of a real-time extension to the π-calculus. In: Proceedings of the International Conference on Formal Techniques for Distributed Systems (FMOODS & FORTE'10), LNCS (2010)
51. Posse, E., Vangheluwe, H.: `kiltera`: a simulation language for timed, dynamic structure systems. In: Proceedings of the 40th Annual Simulation Symposium (ANSS'07) (2007)
52. Ramos, R., Sampaio, A., Mota, A.: A semantics for UML-RT active classes via mapping into Circus. In: Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems FMOODS'05, Volume 3535 of LNCS, pp. 99–114. Springer, Berlin (2005)
53. SAE International: Architecture Analysis & Design Language (AADL). SAE Standard AS5506b, 10 September (2012)
54. Selic, B.: Using UML for modeling complex real-time systems. In: Mueller, F., Bestavros, A. (eds.) Languages, Compilers, and Tools for Embedded Systems (LCTES'98), Volume 1474 of LNCS, pp. 250–260. Springer, Berlin (1998)
55. Selic, B.: Personal Communication, 1 February (2012)
56. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object Oriented Modeling. Wiley, New York (1994)
57. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Whitepaper, Rational Software Corp (1998)
58. Shankar, S., Asa, S.: Formal semantics of UML with real-time constructs. In: UML, Volume 2863 of LNCS, pp. 60–75. Springer, Berlin (2003)
59. Störrle, H., Hausmann, J.H.: Towards a formal semantics of UML 2.0 activities. In: Proceedings German Software Engineering Conference, Volume 65 of LNI, pp. 117–128 (2005)
60. von der Beeck, M.: A structured operational semantics for UML-statecharts. SoSyM **1**(2), 130–141 (2002)
61. von der Beeck, M.: A formal semantics of UML-RT. In: Proceedings of MoDELS'06, pp. 768–782 (2006)
62. Yeung, W.L., Leung, K.R.P.H., Wang, J., Dong, W.: Improvements towards formalizing UML state diagrams in CSP. In: Proceedings of APSEC-'05, pp. 176–184. IEEE Computer Society (2005)
63. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation, 1st edn. Academic Press, New York (1976)
64. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation, 2nd edn. Academic Press, New York (2000)
65. Zhang, T., Huang, S., Huang, H.: An operational semantics for UML RT-statechart in model checking context. In: Proceedings of the 4th International Conference on Internet Computing for Science and Engineering (ICICSE), pp. 12–18 (2009)

Ernesto Posse is a postdoctoral fellow in the Modelling and Analysis in Software Engineering Group (MASE) at the School of Computing at Queen's University. He received M.Sc. and PhD degrees in Computer Science from McGill University in 2001 and 2009. Dr. Posse's interests include formal methods in software engineering with emphasis on real-time embedded systems as well as programming and modelling language tool development.

Juergen Dingel joined the School of Computing at Queen's University in Kingston, Ontario, Canada, in 2000. He received an M.Sc. in Computer Science from Berlin University of Technology in 1992, an M.Sc. in Pure and Applied Logic in 1994 and a PhD in Computer Science in 1999 from Carnegie Mellon University. He is on the editorial board of Software and Systems Modeling (SoSyM) and PC Co-chair of the 17th International Conference on Model Driven Engineering Languages & Systems (MODELS'14). At Queen's University, he leads the Modelling and Analysis in Software Engineering Group (MASE). His research is by funded by IBM, General Motors, Natural Sciences and Engineering Council of Canada (NSERC), and the Ontario Centres of Excellence (OCE). Dr. Dingel's research interests span many aspects of software engineering and formal methods, including software modelling, model-driven engineering, analysis of software artifacts, formal specification and verification, testing, and software quality assurance.