Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2009, Article ID 656810, 26 pages
doi:10.1155/2009/656810

Research Article

A Polyadic pi-Calculus Approach for the Formal Specification of UML-RT

J. M. Bezerra and C. M. Hirata

Computer Science Department, Instituto Tecnológico de Aeronáutica (ITA), São José dos Campos, SP 12228-900, Brazil

Correspondence should be addressed to J. M. Bezerra, juliana@ita.br

Received 6 November 2008; Revised 31 March 2009; Accepted 20 May 2009

Recommended by Thomas B. Hilburn

UML-RT is a UML real-time profile that allows modeling event-driven and distributed systems; however it is not a formal specification language. This paper proposes a formal approach for UML-RT through a mapping of the UML-RT communicating elements into the π-calculus (or pi-calculus) process algebra. The formal approach both captures the intended behavior of the system being modeled and provides a rigorous and nonambiguous system description. Our proposal differentiates from other research work because we map UML-RT to π-calculus, and we allow the mapping of dynamic reconfiguration of UML-RT unwired ports. We illustrate the usage and applicability of the mapping through three examples. The first example focuses on explaining the mapping; the second one aims to demonstrate the use of the π-calculus definitions to verify system requirements; the third case is an example of mobile processes called Handover protocol.

1. Introduction

The specification and development of real-time systems are a challenge due to their characteristic of criticality and safety. Formal methods have been seen as effective in real-time system development [1, 2]. The main advantage of formal methods is to allow a rigorous and nonambiguous system description; so it is possible, through formal verification, to evaluate some aspects such as consistency, completeness, and correctness. The formal verification involves the formal modeling of the system, the formal specification of requirements or properties, and the inference rules to prove that the model satisfies the properties (model checking) [3–5].

The standard DO-178B, Software Considerations in Airborne Systems and Equipment Certification [6], is an acceptable guideline for embedded software approval used by the certification authorities. It suggests the use of formal methods to complement tests, because they generally increase confidence on correct behavior or that anomalous behavior will not occur. Furthermore, formal verification of models helps the designers to find out errors earlier in the modeling phase, which is essential to reduce costs according to Pressman [7].

The most well-known modeling language is Unified Modeling Language (UML), officially defined by the Object Management Group (OMG). Up to the UML version 1.5 [8], the standard lacks support for some important aspects of embedded real-time systems, such as time constraints, signals, and independent components. Aiming to adapt UML to real-time system modeling, some UML profiles were proposed [9, 10]. For instance, Rational Software Company (now IBM) defined the UML RealTime (UML-RT) [11] profile that permits to model distributed and event-driven systems. The profile is supported by the IBM Rational Rose RealTime (RoseRT) tool [12].

With the advance of UML 2.0 [13], UML was improved to model large-scale software systems including the ability to model entire system architectures. The basis of the improvement comes from the experience with various architectural description languages, such as UML-RT [14]. RoseRT is now commercialized as IBM Rational Rose Technical Developer tool [15]; however the features of UML-RT are still present, because they are the source of UML 2.0 structure concepts, for example, the capsule is directly translated to structured classes in UML 2.0.

As UML-RT is not a formal specification language; it is not possible to formally verify the models that are specified in UML-RT. So some researchers propose the transformation of the UML-RT model to a formal model using process algebras [16–19]. The process algebras to which transformation are proposed include CSP [20], CCS [21], and Circus [22]. Other work [23] represents the UML-RT model as timed automata.

In the present paper, we propose a formal approach to the UML-RT through a mapping of the UML-RT communicating elements into the π-calculus [24]. The π-calculus is a process algebra for systems that communicate concurrently. Its advantage is the ability to model mobility, which is made by passing channels as data through channels. The π-calculus does not address the computations that are not related to communication with other processes.

This work is an extension of our earlier work on mapping from UML-RT to π-calculus [25]. In that work, Bezerra et al. present the mapping to a π-calculus syntax version used by HAL-JACK (HD-Automata Laboratory) [26], which is an integrated tool set for the specification, verification, and analysis of concurrent and distributed systems. In this paper, we present the complete mapping using the polyadic π-calculus. The chosen syntax is the one proposed by Milner [24] and not the other one used by HAL-JACK (HD-Automata Laboratory) [26] tool, which is monoadic. The importance of polyadic is the ability to formally represent the communication of more than one message at a time. Besides, we include mapping rules to reason about the following UML-RT concepts: the entry and the exit actions, the composite states, and the transition chains.

The rest of the paper is organized as follows. In Section 2, we provide an overview of UML-RT and π-calculus, and we also describe the related work. We present the UML-RT to the π-calculus mapping in Section 3, and we illustrate the mapping with three examples in Section 4. The first example is used to explain how to apply the mapping to a UML-RT model and obtain the corresponding π-calculus definitions. The second example is employed to demonstrate that the designer is able to use the π-calculus definitions to verify if the model meets the system requirements. The third example is the Handover protocol presented in [24], and it illustrates the use of polyadic communication. Finally, in Section 5 some conclusions are drawn.

2. Theoretical Foundations

This section starts with the description of the main elements of UML-RT, which are considered in the mapping of π-calculus whose description is presented in what follows. We end the section by discussing the related work.

2.1. UML-RT. UML-RT is an extension of UML with elements to facilitate the design of real-time systems. Three of the elements, *capsule*, *port*, and *connector*, are used to model system structure; one element, *protocol*, models the communication inside the system [11].

A *capsule* is an active class and represents software components that can be concurrent and physically distributed. A capsule has a state diagram and a structure diagram. A state diagram is similar to a UML standard state diagram and describes the capsule behavior. A structure diagram details the internal structure of a capsule, which includes subcapsules and their connections with each other. Figure 1 shows the structure diagram of the capsule *TopSystem* that comprises four subcapsules: *source* (instance of the capsule *Source*), *router* (instance of the capsule *Router*), *target1* (instance of the capsule *ConsumerTarget*), and *target2* (instance of the capsule *ExporterTarget*). The model was constructed using the RoseRT tool.

A *port* permits to exchange messages between capsules. A *protocol* needs to be specified for a port. A protocol defines both the number of participants in the communication and the signals that are received and sent by each participant. The *connectors* act as communicating channels between ports that must play different roles of the same protocol. A line between two ports represents a connector. In Figure 1, the ports *q* and *r* implement the protocol named *ConfigProtocol,* and they are connected to each other. The ports play conjugated roles, which can be noted in their representations: *q* as an empty (blank) square and *r* as a filled (black) square.

A port can be classified in terms of visibility (*public* and *protected*), termination (*end* and *relay*), and connectivity (*wired* and *unwired*). A *public* port is located at the capsule border and can interact with other capsule ports. A *protected* port is not visible from the outside; it serves as the communication point of the capsule with its subcapsules. In Figure 1, the capsule *System* has the public port *out* (indicated by the symbol +), and the protected ports *setCT* and *setET* (indicated by the symbol #).

With respect to the termination, an *end* port provides the access to the capsule behavior given by its state diagram. In Figure 1, *setCT* and *setET* are end ports of the capsule *TopSystem*. A *relay* port is used to export the subcapsule interfaces. When a relay port of a parent capsule receives a message, the message is passed automatically to the connected subcapsule port. And when a subcapsule sends a message through a port connected to a relay port of its parent capsule, the message is directly sent to the outside of the parent capsule. By definition, a relay port is public and wired. In Figure 1, *out* is a relay port of the capsule *TopSystem*.

A *wired* port (e.g., *q* and *r* in Figure 1) has a connector joining them to exchange messages. An *unwired* port does not have a connector with other ports, but it allows dynamic communication during runtime through its name registration. In Figure 1, *alert*, *s*, and *t* are examples of unwired ports. Unwired ports can be registered to receive and send signals. For instance, the port *s* can be registered with the subscriber name *x*, and *t* can be registered with the subscriber name *y*. In this case, if the port *alert* is configured with the provider name *x*, the signal sent by *Router* along *alert* is received by *ConsumerTarget*. Similarly, if the port *alert* is set to *y*, its signal is received by *ExporterTarget*. In RoseRT, the subscriber port is called Service Access Point (SAP) and the provider port is called Service Provisioning Point (SPP).

As mentioned, a capsule has also a state diagram. Figure 2 presents the state diagram of the *ExporterTarget* capsule. The initial point (filled circle) is a point which explicitly shows the

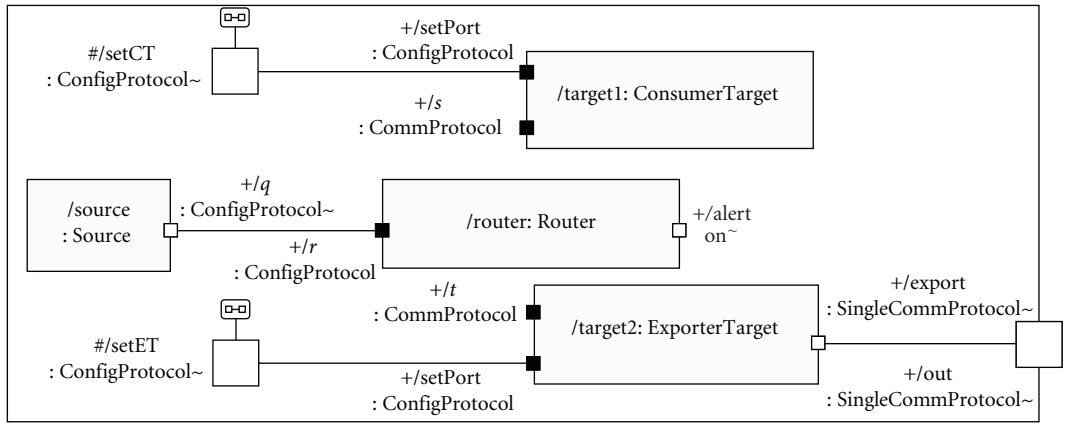


Figure 1: The structure diagram of the capsule *TopSystem*.

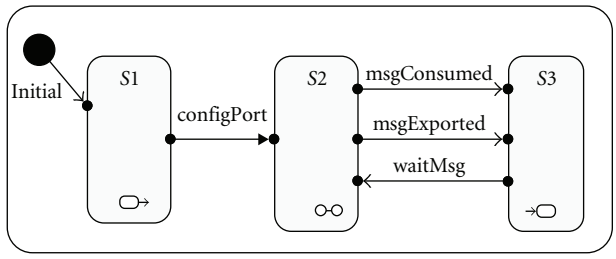


Figure 2: The state diagram of the capsule *ExporterTarget*.

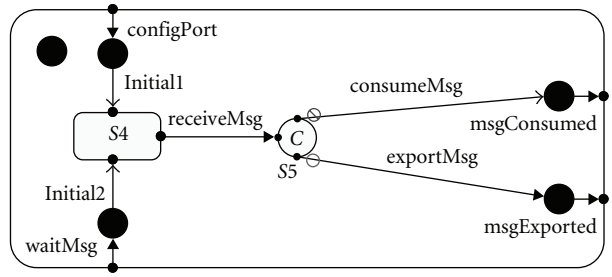


Figure 3: The composite state *S2* of *ExporterTarger* state diagram.

beginning of the state machine. The initial transition, named *Initial*, connects the initial point to the initial state. It does not have an associated trigger; however it may execute an associated action.

A *state* is a condition in which the object is ready to process events. In Figure 2, there are states *S*1, *S*2, and *S*3. We can define actions to be performed when the control enters in a state (*entry* action, indicated by the symbol →▭) and when it leaves a state (*exit* action, indicated by the symbol ▭→). A state that does not contain substates is called a simple state. A state that contains substates is called a composite or hierarchical state (indicated by the symbol ○○). For example, *S*1 and *S*3 are simple states; whereas *S*2 is a composite because it is composed of the simple states *S*4 and *S*5 as illustrated in Figure 3. Furthermore, a state can be a choice point, for example, *S*5, which allows a single transition to be split into two outgoing transition segments, after evaluating a condition on which branch to take. For instance, if the decision evaluation inside *S*5 returns *true*, the transition *exportMsg* is triggered; otherwise, the transition *consumeMsg* is triggered.

A *transition* is a relationship between two states: a source state and a destination state. It specifies the change of control from the source state to the destination state when an object in the source state receives a specified event and some conditions are met. Three concepts related to a transition include: *trigger*, *guard condition,* and *action*. *Trigger* indicates which events (signals) from which interfaces (ports) cause the transition to be taken; however the transition fires only if the *guard condition* is satisfied. After the transition activation, its *action* is executed. Actions are operation calls, a variable handling or a signal dispatch to another capsule.

Transitions can span multiple hierarchies changing context on the way from the source to the destination state. Therefore they must be partitioned into different segments by the junction pointers. Each transition segment has a distinct name. Any segment can execute actions; however only the originating segment has a trigger. For instance, the transition *configPort* illustrated in Figure 2 connects the states *S*1 and *S*2, but it comprises two segments inside the state *S*2 as shown in Figure 3: the transition *configPort* itself and the transition *Initial1* separated by a junction pointer.

It is important to mention that the triggers and the actions are configured internally in the RoseRT tool, and the model screenshot presents only the state and transition names. Table 1 presents the transition information of the *ExporterTarget* state diagram, where the action is specified in C programming language. For example, the transition *configPort* triggers if it receives along the port *setPort* the signal *portName*. The transition action is the storage of the message, received through the port *setPort*, in the local variable *aux* using the statement "*aux* = **rtdata*;". Later the port *t* is renamed to *aux*, using the statement "*t*·*registerSAP(aux);*". The transition *exportMsg* does not have a trigger, but it executes an action of sending along the port *export* the signal *msg* with the message *d*1.

Table 1: Transitions of the *ExporterTarget* state diagram.

Transition	Trigger	Action
Initial	—	—
configPort	Port: *setPort*	aux = *rtdata;
	Signal: *portName*	t·registerSAP(aux);
Initial1	—	—
Initial2	—	—
receiveMsg	Port: t	m = *rtdata;
		$d1 = m$·msg1;
	Signal: *msg*	$d2 = m$·msg2;
consumeMsg	—	—
exportMsg	—	export·msg($d1$)·send();
msgConsumed	—	—
msgExported	—	—
waitMsg	Port: *timer*	—
	Signal: *timeout*	

UML-RT allows defining a class and sending an object of this class as a message. For example, the capsule *Exported-Target* receives a message m in the transition *receiveMsg* with the attributes *msg1* and *msg2*, which are stored, respectively, in $d1$ and $d2$ variables, according to Table 1. Later, by the transition *exportMsg*, the capsule *ExportedTarget* is able to send the message $d1$. However, the message m can be seen as two messages $m1$ and $m2$ transmitted in a single signal, which characterizes the communication of more than one message at a time in UML-RT.

Although, there are other diagrams available in the UML-RT, such as use case, component, deployment, class, collaboration, and sequence diagrams; the proposed mapping rules deal only with the state and structure diagrams of UML-RT capsules.

A use case diagram is specially used before the modeling phase in order to identify the functionality of the system. A component diagram shows the dependencies among software components that exist at compilation time, linking time, or run-time. A deployment diagram captures the physical distribution of the run-time processes across a set of processing nodes. As our scope is focused on design, and not on the requirement or implementation, the use case, component, and deployment diagrams are not addressed.

A class diagram shows the static structure of the model. It may contain other elements besides classes, such as capsules and protocols. A collaboration diagram captures a desirable pattern of interactions between objects, emphasizing the structural organization of the objects. A capsule structure diagram is a specialized form of the collaboration diagram and includes information available also in the class diagram. A sequence diagram specifies communication scenarios of collaboration; however with the capsule structure diagram and the state diagram of each capsule it is possible to obtain all the possible collaborations; therefore the sequence diagram is not used. So our mapping considers only the state and structure diagrams of UML-RT capsules.

2.2. The π-Calculus. The π-calculus [24] is a process algebra for systems that communicate concurrently, that is, systems composed of processes that run in parallel and interact through channels. The main difference between π-calculus and its predecessors, CSP and CCS, is the possibility to pass channels as data through channels [27–29]. This characteristic allows the π-calculus to express mobility.

The prefix π of the π-calculus represents $\overline{x}\langle y\rangle$, $x(y)$, or the unobservable action τ. The term $\overline{x}\langle y\rangle$ indicates that the π-calculus port (or channel) x sends the π-calculus message y. The term $x(y)$ indicates that the message y is received along the port x. The round brackets (y) are used for the binding occurrence of a parametric name (one that may be instantiated by another name), and the angle brackets $\langle y\rangle$ for nonbinding occurrences of a name.

The π-calculus process expression P is defined by the syntax:

$$P := \sum_{i\in I} \pi_i \cdot P_i \mid P_1 \mid P_2 \mid new\ a\, P \mid !P. \tag{1}$$

The term $\sum_{i\in I} \pi_i \cdot P_i$ indicates summation, where I is a finite indexing set. The dot is the sequence operation, which means that π_i will occur before P_i becomes activated. If $i = 2$, we have $\pi_1 \cdot P_1 + \pi_2 \cdot P_2$. If $i = 0$, we have $P = 0$, that is a stop process. In general, we omit the stop process, for example, we write $x(y)$ instead of $x(y) \cdot 0$. The parallel composition $P_1 \mid P_2$ means that P_1 and P_2 run concurrently. The term *new* $a\,P$ represents the restriction of the name a in the P context, that is, the name a is bound in P and it is not seen outside. The replication operator $!P$ is the composition of unlimited copies of P.

For example, considering the following processes of B, C, D, and E, where

$$B = (\overline{z}\langle x\rangle + \overline{z}\langle y\rangle), \tag{2}$$

$$C = z(a) \cdot \overline{a}\langle b\rangle, \tag{3}$$

$$D = x(c), \tag{4}$$

$$E = y(d). \tag{5}$$

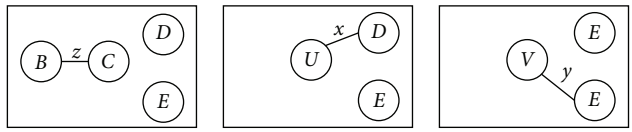
The process B is able to send the message x along the port z or to send the message y along the port z. The process C can receive a message a through the port z, and later send the message b through the port a. The process D can only receive a message c along the port x. Finally, the process E is able to receive a message d via the port y.

Assuming that the process A (Figure 4(a)) consists of B, C, D, and E above, so by using π-calculus, the process A is the parallel composition of the four processes:

$$A = B \mid C \mid D \mid E. \tag{6}$$

We can replace in (6) the definition of each process using (2) to (5), so

$$A = (\overline{z}\langle x\rangle + \overline{z}\langle y\rangle) \mid z(a) \cdot \overline{a}\langle b\rangle \mid x(c) \mid y(d). \tag{7}$$



(a) (b) (c)

Figure 4: The π-calculus mobility. (a) A process, (b) $A1$ process, and (c) $A2$ process.

To understand how to manipulate the definitions included in the process A definition, it is necessary to understand the π-calculus concepts related to the structural congruence and the reactions.

The processes P and Q in the π-calculus are structurally congruent, written $P \equiv Q$, if one can be transformed into the other using the rules.

(i) Change of Bound Names (Alpha-Conversion). The alpha-conversion allows bound variable names to be renamed. As an example, in (2), we have that $D = x(c)$. If $D' = x(c)$, so $D \equiv D'$, that is, D is the same as D'.

(ii) Reordering Terms in a Summation. The reordering terms in a summation are related to the commutative operations. For instance, in (1), we have that $B = (\overline{z}\langle x\rangle + \overline{z}\langle y\rangle)$. But we can rewrite B by reordering its terms, so $B = (\overline{z}\langle y\rangle + \overline{z}\langle x\rangle)$.

(iii) $P \mid 0 \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$. The first term indicates that the stop process is the neutral element in the parallel composition. The second and the third terms are the commutative and the associative operation, respectively, of the parallel composition. For example, the definition of A in (6) can also be written by reordering B and C, as $A = C|B|D|E$.

(iv) new $x(P \mid Q) \equiv P \mid$ *new* $x\,Q$, *If* x *Is not a Free Name of* P. It states that if a process P does not have the free name x, the restriction of x in the parallel composition $P \mid Q$ is kept only in Q.

(v) new $x\,0 \equiv 0$, *and new* $x, y\,P \equiv$ *new* $y, x\,P$. The first term explains that there is no need to use restriction in the stop process. The second term indicates that the order of the names in a restriction operation is not important.

(vi) $!P \equiv P \mid !P$. It states that the replication of P is the same as a parallel composition of an infinite number of P.

The reaction relation over π-calculus explains how the processes react with each other. It contains exactly the transitions which can be inferred from the following rules.

(i) TAU $[\tau \cdot P + M \rightarrow P]$. It states that in a process $\tau \cdot P + M$, if τ occurs, then we choose the first option of the choice operator and the result is only the P process.

(ii) REACT $[(x(y) \cdot P + M) \mid (\overline{x}\langle z\rangle \cdot Q + N) \rightarrow \{z/y\}P \mid Q$, *Where* $\{z/y\}P$ *Means that* y *Must Be Replaced by* $z]$. It states that the two processes running concurrently $(x(y) \cdot P + M)$ and $(\overline{x}\langle z\rangle \cdot Q + N)$ may react, because the second process sends the message z along the port x, while the first process receives a message y along the port x. After the reaction, all occurrences of y in the first process are renamed by z.

(iv) PAR [*If* $P \rightarrow P'$, *Then* $P \mid Q \rightarrow P' \mid Q$]. It states that in a parallel composition, if one process transits to another, then the parallel composition is maintained.

(v) RES [*If* $P \rightarrow P'$, *Then new* $xP \rightarrow$ *new* xP']. It states that in a restriction operation, if one process transits to another, then the restriction is maintained.

(vi) STRUCT $[P \rightarrow P'$ *Implies* $Q \rightarrow Q'$, *if* $P \equiv Q$, $P' \equiv Q']$. It states that if P transits to P' it implies that Q transits to Q', then P and Q are structurally congruent.

In the example of the process A defined in (7), the actions $\overline{z}\langle x\rangle$ of B and $z(a)$ of C are complementary and may interact generating $A1$:

$$A1 = 0 \mid \overline{x}\langle b\rangle \mid x(c) \mid y(d). \tag{8}$$

After the reaction $A \rightarrow A1$, all occurrences of a in C are replaced by x. Figure 4(b) shows $A1$ process, where $U = \overline{x}\langle b\rangle$ and B is not represented because it is a stop process.

Additionally, there is another possible interaction in A between $\overline{z}\langle y\rangle$ of B and $z(a)$ of C, generating $A2$:

$$A2 = 0 \mid \overline{y}\langle b\rangle \mid x(c) \mid y(d). \tag{9}$$

After the reaction $A \rightarrow A2$, all occurrences of a in C are replaced by y. Figure 4(c) depicts $A2$ process, where $V = \overline{y}\langle b\rangle$ and B is not represented because it is a stop process.

The example illustrates the π-calculus mobility concept, because the channels x (in the reaction $A \rightarrow A1$) and y (in the reaction $A \rightarrow A2$) are passed as message along the channel z, which links processes B and C. The mobility is the main aspect of the π-calculus whose representation is addressed in this paper.

The presented π-calculus version is the monoadic π-calculus, in which a message contains exactly one name. The polyadic π-calculus [24] permits to exchange messages consisting of more than one name, using the prefixes $x(\vec{y})$ and $\overline{x}\langle\vec{z}\rangle$. In particular, it admits the cases $x()$ and $\overline{x}\langle\rangle$ where $\vec{y}$ and $\vec{z}$ are empty, and we can write as x and $\overline{x}$ to indicate only the synchronization between the ports.

For an example of polyadic π-calculus, consider the following process:

$$E = x(y_1, y_2) \cdot F \mid \overline{x}\langle z_1, z_2\rangle \cdot G \mid \overline{x}\langle w_1, w_2\rangle \cdot H. \tag{10}$$

The first term in (10) may react with the second or the third term along the port x. The reaction with the second term replaces y_1, y_2 by z_1, z_2, respectively. While the reaction with the third term replaces y_1, y_2 by w_1, w_2, respectively.

The definition of E is different from E' written in monoadic π-calculus, as follows:

$$E' = x(y_1) \cdot x(y_2) \cdot F \mid \overline{x}\langle z_1\rangle \cdot \overline{x}\langle z_2\rangle \cdot G \mid \ \overline{x}\langle w_1\rangle \cdot \overline{x}\langle w_2\rangle H. \tag{11}$$

In (11), the first term may react with the second and later with the third term, resulting that y_1 is replaced by z_1, and y_2 by w_2. It is also possible that the first term may react with the third and later with the second term. In both cases, the result of (11) using monoadic π-calculus is different from that one expected in (10) using polyadic π-calculus. For the correct encoding of the polyadic using monoadic, it is necessary to guarantee that there can be no interference on the channel along which a composite message is sent. It is addressed by Milner in [24].

A process in monoadic or polyadic π-calculus can also have a list of input names written between round brackets, for example,

$$I(m,n) = \overline{m}\langle n\rangle \cdot I'. \tag{12}$$

In this case, the definition of process I has the port name m and the message n available as input parameters. If a process J uses the definition of I, then J calls I definition passing the input names between angle brackets, for example,

$$J(p) = p(q) \cdot I\langle q,r\rangle. \tag{13}$$

In (13), J references I definition passing q and r to, respectively, rename I inputs m and n.

So in π-calculus, the round brackets () are used to both organize the input set of processes and organize the set of messages received through a port. Whereas the angle brackets $\langle\rangle$ are used to organize both the input set of process references and the set of messages sent through a port.

As described above, the π-calculus is a process algebra to describe and analyze concurrent systems consisting of processes (or agents) which interact with other connected processes.

2.3. Related Work. Some approaches use Z [30] and Object-Z [31] to formalize UML models. Miao et al. [32] deal with the UML class, sequence, and statechart diagrams; while Kim and Carrington [33] present a formal Object-Z model of the UML state machine. However, they consider UML and not UML-RT with its architectural components. That is because Z and Object-Z are suitable for capturing data and states, and not for capturing dynamic communication configurations. On the other hand, π-calculus is suitable to support dynamic communication, that is the focus of our UML-RT formalization proposal. Due to the π-calculus contribution, other formalisms propose the combination of a state-based formalism and a dynamic action-based calculus by integrating the mobility concepts of the π-calculus to Z and Object-Z [34–36].

Terriza et al. [16] describe a mapping of UML-RT to CSP + T, whose objective is to provide time elements to UML-RT, making the mapping of the capsule state diagram and the class diagram to CSP + T definitions. CSP + T is a formal specification language that adds time interval description to CSP. Fischer et al. [17] propose the conversion of UML-RT structure diagram to CSP process algebra. Engels et al. [18] describe a translation from the UML-RT capsule state and structure diagrams to CSP. Ramos et al. [19] propose a mapping of UML-RT state, structure, and class diagrams to Circus, which is a formal method that combines concepts of CSP and Z [30].

The aforementioned related work deals with synchronous methods. In general, the information in the structure diagram is also provided in the class diagram, and then some authors use one or another. The capsule definition reuse is a criterion that indicates whether the generated definitions in the process algebra can reuse the definition of the capsules. For example, assuming that the capsule A has two subcapsules of type B, the definition of A in π-calculus references twice the B definition, but for each one, A sets different names as input due to the π-calculus mobility. In CSP and CCS, it would be necessary to write two equations of B, one for each configuration that B may have. An example of this in CSP is the traffic light system in [17].

Knapp et al. [23] compile UML-RT state diagram into timed automata and represent the timed-annotated UML collaborations as another timed automaton, using a prototype called HUGO/RT. Afterwards, the prototype calls the model checker UPPAAL to verify the model against the scenario specified by the UML collaboration. Some interesting characteristics include the representation of time considered in a UML-RT timeout and the mapping of the event queue that holds the events not already handled by the machine. However, there are two limitations of this approach. First, the events cannot transmit messages; second, the composite and the choice states are not represented.

Additionally, all the above related work considers only the wired ports in their mappings; that is, they do not map the UML-RT unwired ports, so mobility is not addressed.

In our previous work, we propose the UML-RT to π-calculus mapping rules using the π-calculus syntax version accepted by the HAL-JACK tool. The HAL-JACK syntax uses the monoadic π-calculus, which lead us to represent only the communication of one message at a time. Besides, the HAL-JACK syntax requires that all names used in a definition must be declared in its input list; this restriction causes complex mapping rules. In our proposal, we use the π-calculus syntax proposed by Milner and we represent the communication of more than one message through polyadic π-caculus. Other contribution comprises the mapping of the entry and the exit actions, the composite states, and the transition chains.

3. The UML-RT to Polyadic π-Calculus Mapping

In this section, we propose the mapping from the UML-RT to the π-calculus. We make some initial considerations regarding the mapping, and later we explain the mappings of the following elements of UML-RT: capsule, structure diagram, state diagram, state, and transition.

3.1. Initial Considerations. The processing of a single event at a time by a state machine is known as a run-to-completion (RTC) step, which means that a transition cannot be interrupted by the arrival of an event. When the transition is partitioned into different segments, in the case of the composite states, only the originating segment has a trigger defined, so there is only one event to be processed. So, even in this case, the transition chain (i.e., the sum of all transition segments) is executed in one RTC step. The communication model used by the π-calculus is the synchronized communication, which is suitable to the RTC step, because the object corresponding to the sender state diagram blocks until it receives the notification about the receipt of the event at the object corresponding to the receiver state diagram.

The polyadic π-calculus definitions resulted from the mapping follow the syntax proposed by Milner [24]. The UML-RT capsule state diagrams are used in the mapping to retrieve the behavior of the capsules, while the structure diagrams are used to retrieve the association between capsules. The base of the mapping is that a UML-RT port is represented by a π-calculus port; whereas the messages transmitted through the signal are represented by the π-calculus messages. In order to be able to represent the transmission of zero or more messages, our approach uses the polyadic π-calculus.

UML-RT offers a limited support for time annotations, there is one element to set a timer and simulate a timeout. In the RoseRT tool, this timer is allowed through a port that implements a built-in *Timing* protocol. In our mapping, the timeout is represented by the unobservable action τ, because it is a private interaction inside the capsule.

The main elements of a transition are trigger, action, and guard condition. However, as the focus of the UML-RT to π-calculus mapping is on communication elements, guard condition is not considered. The transition is triggered by the receipt of a message. The considered transition actions are a message sending or a name reconfiguration, where a name reconfiguration is a way to reconfigure the unwired ports. We assume that the entry and exit actions are only the message sending statements.

The RoseRT tool allows the use of three programming languages, C, C++, and Java, to specify the transition and the state actions. Aiming to be independent from programming languages and simplify the notation used in the mapping rules, we propose the following pseudo codes.

(i) The message receipt statement is written as "p *receive* $e(\vec{m})$", which means that the port p receives the signal e with the list $\vec{m}$ of messages.

(ii) The message sending statement is represented as "p *send* $e(\vec{m})$", that is, the port p sends the signal e with the list $\vec{m}$ of messages.

(iii) The name reconfiguration statement is written as "*name1* = *name2*", that is, *name1* is from now on used with the value of *name2*. In the RoseRT, this information is given through the commands SAP or SPP, for example, "$name1 \cdot registerSAP(name2)$".

It is important to recall that the list $\vec{m}$ of messages can be empty, when there exists only the synchronization between the ports and no message is transmitted.

Some notations used in the mapping rules need to be presented.

(i) The term $P(L)$ represents the π-calculus definition of P with its input set L.

(ii) The term $P\langle L\rangle$ represents a reference to the π-calculus definition of P. The input set L is passed to the P definition. This notation is used in the syntax proposed by Milner and is commented in Section 2.2.

(iii) The term P represents the π-calculus definition of a component without a defined input set. Each π-calculus reference to this component should be replaced by the definition P.

(iv) The result of $L_1 - L_2$ is L_1 without the elements contained in L_2, where L_1 and L_2 are sets.

(v) The symbol $\bigcup$ is the union operator in sets. So, $\bigcup_{i\in N} L_i = L_1 \cup L_2 \cup \cdots \cup L_N$.

(vi) The symbol $\sum$ indicates the application of the choice operator in π-calculus processes. So, $\sum_{i\in N} P_i = P_1 + P_2 + \cdots + P_N$.

(vii) The symbol $\prod$ indicates the application of the sequence operator in π-calculus actions. So, $\prod_{i\in N} \pi_i = \pi_1 \cdot \pi_2 \cdots \pi_N$.

The result of UML-RT mapping is the π-calculus definition of the main capsule in UML-RT model. After introducing the essence of the mapping and the notations, we are able to present the capsule mapping.

3.2. Capsule Mapping. Each UML-RT capsule has its own state diagram and may contain other capsules in its structure diagram. So, the capsule definition in the π-calculus is composed by two parts: the structure diagram definition in the π-calculus and the state diagram definition in the π-calculus. These two definitions are arranged with the parallel composition operator of the π-calculus, as presented in Definition 1, because the behavior of the capsule and its subcapsules are executed concurrently in UML-RT.

Definition 1 (Capsule definition). Given a capsule P, its π-calculus definition is the parallel composition of its state diagram definition and its structure diagram definition. So, the π-calculus definition of the capsule P is given by

$$\begin{aligned} &P(InputSet[P]) \\ &\quad = new(RestrictedSet[P]) \\ &\qquad \cdot (P\_StructureDiagram \mid P\_StateDiagram), \end{aligned} \tag{14}$$

where

(i) $InputSet[P]$ is the input message set of the P capsule definition. It is defined as the set of the public wired ports of the capsule P.

(ii) $RestrictedSet[P]$ is the set of messages restricted to the context of the capsule P. Assuming that P has N subcapsules Q_i, the restricted set is defined as

$$RestrictedSet[P] = \bigcup_{i \in N} (InputSet[P, Q_i]) - InputSet[P]. \tag{15}$$

(iii) $InputSet[P, Q_i]$: input set used by the parent capsule P to reference the definition of its subcapsule Q_i. It is defined in the structure diagram mapping.

In the example presented in Figure 1, using Definition 1, the main capsule is *TopSystem* and its π-calculus definition can be written as

$$\begin{aligned} &TopSystem(InputSet[TopSystem]) \\ &\quad = new(RestrictedSet[TopSystem]) \\ &(TopSystem\_StructureDiagram \\ &\quad \mid TopSystem\_StateDiagram) \\ &InputSet[TopSystem] = \{out\} \\ &RestrictedSet[TopSystem] \\ &\quad = \{InputSet[TopSystem, Source] \\ &\quad \cup InputSet[TopSystem, Router] \\ &\quad \cup InputSet[TopSystem, ConsumerTarget] \\ &\quad \cup InputSet[TopSystem, ExportedTarget]\} \\ &\quad - \{out\}. \end{aligned} \tag{16}$$

In order to specify the capsule definition, the next sections explain how to obtain the π-calculus definitions for the structure diagram and the state diagram.

3.3. Structure Diagram Mapping. A capsule may include subcapsules in its structure diagram. If the structure diagram of a capsule does not have subcapsules, this diagram is defined in the π-calculus as the stop process. In this case, the capsule definition is composed only by the state diagram definition, because the parallel composition between a process and a stop process is the first process. However, if the capsule has subcapsules, its structure diagram definition is the π-calculus parallel composition of the subcapsules' definitions, as explained in Definition 2, because all UML-RT subcapsules execute concurrently. In this case, the definition of each subcapsule has to be written using Definition 1 and the connections between ports are used to specify how the parent capsule references its subcapsules.

Definition 2 (Structure diagram definition). Given a capsule P. If P does not have subcapsules, the π-calculus definition of the P structure diagram is the stop process:

$$P\_StructureDiagram = 0. \tag{17}$$

Otherwise, if P has subcapsules, given a set N of subcapsules Q_i of P. Given the π-calculus definition of each subcapsule Q_i as $Q_i(InputSet[Q_i.\ ])$, the π-calculus definition of the P structure diagram is given by

$$\begin{aligned} &P\_StructureDiagram \\ &\quad = Q_1\langle InputSet[P, Q_1]\rangle \mid \cdots \mid Q_N\langle InputSet[P, Q_N]\rangle, \end{aligned} \tag{18}$$

where the $InputSet[P, Q_1]$ is defined based on the connection between the wired ports as follows.

(a) If the port p of the parent capsule P is connected to the public port q_1 of the subcapsule Q_1, the port q is renamed to p. So, $p \in InputSet[P, Q_i]$.

(b) If the public port q_1 of the subcapsule Q_1 is connected to the public port q_2 of the subcapsule Q_2, a new name z is generated dynamically to rename the ports q_1 and q_2. So,

$$\begin{aligned} &z \in InputSet[P, Q_1], \\ &z \in InputSet[P, Q_2]. \end{aligned} \tag{19}$$

In the example shown in Figure 1, according to Definition 2, the π-calculus definition of *TopSystem* structure diagram is the parallel composition of the definition of the subcapsules: *Source*, *Router*, *ConsumerTarget,* and *ExporterTarget*. Considering the input sets used by *TopSystem* to reference its subcapsules, we have that the port *setPort* of *ConsumerTarget* is renamed to parent port *setCT*, the port *setPort* of *ExporterTarget* is renamed to parent port *setET*, and the new name z is created to rename the q_1 and r_1. So, *TopSystem* structure diagram is defined as

$$\begin{aligned} &TopSystem\_StructureDiagram \\ &\quad = Source\langle z\rangle \mid Router\langle z\rangle \mid ConsumerTarget\langle setCT\rangle \\ &\quad \mid ExporterTarget\langle setET, out\rangle. \end{aligned} \tag{20}$$

In order to complete the capsule definition in π-calculus, the next section presents how to obtain the state diagram definition.

3.4. State Diagram Mapping. A capsule can specify or not an associated state diagram. If it is not specified, the π-calculus definition of the capsule state diagram is the stop process. Otherwise, if the capsule has a state diagram, it is composed by states and transitions between them. The first transition to be activated is the initial transition, which is connected to the initial state. As explained in Section 2.1, the initial transition does not have a trigger, it runs automatically when the state diagram initializes, but it can execute an action. So, the π-calculus definition of the state diagram is the sequence composition between the definition of initial transition action and the definition of the initial state, as explained in Definition 3.

Definition 3 (State diagram definition). Given a capsule P. If P does not have a state diagram, the π-calculus definition of the state diagram is the stop process:

$$P\_StateDiagram = 0. \tag{21}$$

Otherwise, if P has a state diagram, assuming that *Initial-Transition* is the initial transition and *S1* is the initial state, the π-calculus definition of the state diagram is given by

$$P\_StateDiagram = P\_\ InitialTransition\_Action \cdot P\_S1\langle InputSet[P, InitialPoint, S1]\rangle. \tag{22}$$

where $InputSet[P, InitialPoint, S1]$ is the input message set used by the initial transition definition to reference the definition of the state $S1$. It is defined in the transition action mapping.

The definition of the initial transition action is explained in the transition action mapping, and the definition of the initial state is explained in the state mapping. Note that the initial transition trigger is not mentioned in Definition 3, because the initial transition does not have a trigger, it runs automatically when the state diagram initializes. The definitions of the other states reachable from the initial state, which are included in the capsule state diagram, are considered in the definition of the initial state.

In the example presented in Figure 2, the *ExporterTarget* state diagram is defined, using Definition 3 as

$$\begin{aligned} &ExporterTarget\_StateDiagram \\ &\quad = ExporterTarget\_Initial\_Action \\ &\qquad \cdot ExporterTarget\_S1 \\ &\qquad \cdot \langle InputSet[ExporterTarget, InitialPoint, S1]\rangle. \end{aligned} \tag{23}$$

According to Definition 3, the state diagram definition references the initial state definition, which depends on the other states and transitions in the state diagram. So, the next section details the state and transition mapping.

3.5. State Mapping. The name of a UML-RT state in the π-calculus definition is specified as the concatenation of the UML-RT capsule name and the UML-RT state name to guarantee unique definitions in π-calculus. An entry action is executed whenever the state is entered; regardless of which incoming transition is taken. Besides, a state has multiple outgoing transitions which can be taken. So, the π-calculus definition of the UML-RT state, as explained in Definition 4, is the sequence composition between its entry action definition and the choice composition of each transition definition.

Definition 4 (Simple state definition). Given a simple state S in the state diagram of a capsule P, and given $InputSet[P,S]$. Assuming that the state S has an entry action and a set T of outgoing transitions, the π-calculus definition of the state S is given by

$$P\_S(InputSet[P, S]) = P\_S\_EntryAction \cdot \sum_{i\in T}(P\_S\_OutgoingTransition_i), \tag{24}$$

where $InputSet[P,S]$ is the input message set of the S definition. It is defined in the transition action mapping.

For instance, in Figure 2, the $S1$ definition can be written using Definition 4 as

$$\begin{aligned} &ExporterTarget\_S1(InputSet[ExporterTarget, S1]) \\ &\quad = ExporterTarget\_S1\_EntryAction \\ &\qquad \cdot P\_S1\_configPort. \end{aligned} \tag{25}$$

The UML-RT choice point is a special case of UML-RT state, where there is no entry and exit action, and the transition does not have a trigger, because it is enabled according to the condition specified in the choice point. So, Definition 4 can be used to provide the π-calculus definition of a choice point too.

According to Definition 4, the entry action definition is a part of the state definition. The exit action definition is used only in the transition definition. The entry (or exit) action may not exist in a UML-RT state, so its reference is omitted from the state (or transition) definition. Whether the state performs an entry or exit actions, they include message sending actions in UML-RT, then the π-calculus definition of the entry and exit actions are the sequence composition of the message sending actions. The definition of a state action is provided in Definition 5 and may represent the definition of an entry action or an exit action.

Definition 5 (State action definition). Given a state S in the state diagram of a capsule P, the state action can be an entry action and an exit action.

If the state action is not defined for S, the π-calculus reference to the state action definition is omitted from the parent definition.

Otherwise, if the state action is specified as a set N of message sending actions "p_i *send* $e_i(\vec{m}_i)$", the π-calculus definition of the state action is given by

$$P\_S\_StateAction = \prod_{i\in N}\left(\overline{p}_i\langle\vec{m}_i\rangle\right). \tag{26}$$

After presenting the state mapping, the next section explains the transition mapping.

3.6. Transition Mapping. In a UML-RT model, the transition has an associated trigger and may execute an action after being activated. Considering the state with its outgoing transitions, it is important to mention that the state exit action is taken whenever the control leaves the state from whatever outgoing transition; so the state exit action has

to be computed after the transition trigger and before the transition action. Then, the outgoing transition definition in the π-calculus, as detailed in Definition 6, is a sequence composition of the transition trigger definition, the state exit action, the transition action definition, and the target state definition.

Definition 6 (Outgoing transition definition). Given the simple states $S1$ and $S2$ in the state diagram of a capsule P. Given an outgoing transition from $S1$ to $S2$, the π-calculus definition of the transition is

$$\begin{aligned} &P\_S1\_OutgoingTransition \\ &\quad = P\_S1\_OutgoingTransition\_Trigger \\ &\qquad \cdot P\_S1\_ExitAction \\ &\qquad \cdot P\_S1\_OutgoingTransition\_Action \\ &\qquad \cdot P\_S2\langle InputSet[P, S1, S2]\rangle, \end{aligned} \tag{27}$$

where $InputSet[P, S1, S2]$ is the input message set used by the state $S1$ to reference the definition of the state $S2$. It is defined in the transition action mapping.

For instance, the transition *receiveMsg* can be defined, using Definition 6, as

$$\begin{aligned} &ExporterTarget\_S4\_receiveMsg \\ &= ExporterTarget\_S4\_receiveMsg\_Trigger \\ &\cdot ExporterTarget\_S4\_ExitAction \\ &\cdot ExporterTarget\_S4\_receiveMsg\_Action \\ &\cdot ExporterTarget\_S5\langle InputSet[ExporterTarget, S4, S5]\rangle. \end{aligned} \tag{28}$$

A special case of a UML-RT transition is when the transition is a transition chain that transposes the boundaries of composite states. As it is explained in Section 2.1, this transition is partitioned into different segments by the junction pointers, and only the originating segment has a trigger defined, while all segments can execute actions. In this case, we have to compute in the right order the following aspects: the trigger of the first segment, the actions of all segments, the entry or the exit action of all the transposed parent states, as detailed in Definition 7.

Definition 7 (Definition of the outgoing transition chain). Given an outgoing transition chain from a state $S1$ to a state $S2$, where $S1$ and $S2$ belong to the state diagram of a capsule P. Assuming that the outgoing transition chain transposes the boundary of a set N of the parent states $CS1$'s of $S1$, and later transposes the boundary of a set M of the parent states $CS2$'s of $S2$, where N or M can be empty sets, the transition chain is composed by $N+M+1$ segments. In this case, the π-calculus definition of the outgoing transition chain is given by

$$\begin{aligned} &P\_S1\_OutgoingTransitionChain \\ &\quad = P\_S1\_Segment_1\_Trigger \\ &\qquad \cdot P\_S1\_ExitAction \\ &\qquad \cdot P\_S1\_Segment_1\_Action \\ &\qquad \cdot (P\_S1\_OutgoingTransitionChain\_Up \\ &\qquad\quad \cdot P\_S1\_OutgoingTransitionChain\_Down) \\ &\qquad \cdot P\_S2\langle InputSet[P, S, S2]\rangle, \end{aligned} \tag{29}$$

where

$$\begin{aligned} &P\_S1\_OutgoingTransitionChain\_Up \\ &\quad = \prod_{i\in N} (P\_CS1_i\_ExitAction \cdot P\_S1\_Segment_{i+1}\_Action) \end{aligned}$$

$$\begin{aligned} &P\_S1\_OutgoingTransitionChain\_Down \\ &\quad = \prod_{j\in M} \Big(P\_CS2_j\_EntryAction \\ &\qquad\quad \cdot P\_S1\_Segment_{j+1+N}\_Action\Big). \end{aligned} \tag{30}$$

The *ExporterTarget* state diagram, illustrated in Figures 2 and 3, has the transition chain initiating with the segment *consumeMsg*. This transition chain originates at the state $S5$ and transposes the boundary of the composite state $S2$ until reaching the state $S3$. According to Definition 7, the transition chain *consumeMsg* is defined as

$$\begin{aligned} &ExportedTarget\_S5\_consumeMsg \\ &= ExportedTarget\_S5\_consumeMsg\_Trigger \\ &\cdot ExportedTarget\_S5\_ExitAction \\ &\cdot ExportedTarget\_S5\_consumeMsg\_Action \\ &\cdot ExportedTarget\_S2\_ExitAction \\ &\cdot ExportedTarget\_S5\_msgConsumed\_Action \\ &\cdot ExportedTarget\_S3\langle InputSet[ExporterTarget, S5, S3]\rangle. \end{aligned} \tag{31}$$

As commented in Section 3.1, the transition trigger is a message receipt statement and the transition action can be a message sending statement or a name reconfiguration statement. The transition trigger definition is provided in Definition 8, while the transition action definition is explained in Definition 9.

Definition 8 (Transition trigger definition). Given a transition from state $S1$ to state $S2$ in the state diagram of a capsule P.

If there is no trigger specified to the transition, the π-calculus reference to the transition trigger definition is omitted from the parent definition.

Otherwise, assuming that the transition can be triggered by a set N of message receipt statements as "p_i *receive* $e_i(\vec{m}_i)$", the π-calculus definition of the transition trigger is given by

$$P\_S1\_Transition\_Trigger = \sum_{i \in N} (p_i(\vec{m}_i)). \quad (32)$$

A special case occurs when e_i is a timeout signal. In this case, the trigger is represented by the unobservable action τ. So, $p_i(\vec{m}_i) = \tau$.

Analyzing Definition 8, the transition trigger definition considers all the messages that the capsule can receive in the current transition. However, it cannot be defined, for example, in an outgoing transition of a choice point and in the first segment of a transition chain.

In Definition 9, the transition action considers all the message sending and the reconfiguration statements. In case of a reconfiguration statement, it is important that the old name is the one used by the next state, while the new name is used to reference the next state definition. The input message set of the capsule definition is maintained in the other input sets to keep the consistence of the π-calculus definitions. Note that Definition 9 holds to the actions of all segments in a transition chain.

Definition 9 (Transition action definition). Given a transition from state $S1$ to state $S2$ in the state diagram of a capsule P.

If there is no action associated to the transition, the π-calculus reference to the transition action definition is omitted from the parent definition.

Otherwise, assuming that the transition action is composed of a set N of message sending statements as "p_i *send* $e_i(\vec{m}_i)$" and also by a set M of name reconfiguration statements as "$y_j = x_j$", the π-calculus definition of the transition action is given by

$$P\_S1\_Transition\_Trigger = \prod_{i \in N} \left(\overline{p}_i \langle \vec{m}_i \rangle\right),$$

$$InputSet[P, S1, S2] = \bigcup_{j \in M} x_j \cup InputSet[P], \quad (33)$$

$$InputSet[P, S2] = \bigcup_{j \in M} y_j \cup InputSet[P],$$

where $InputSet[P]$ is the input message set of the P capsule definition, already defined in Definition 1.

For instance, the transition *receiveMsg* of the *ExporterTarget* state diagram, illustrated in Figure 3, has a trigger which can be defined using Definition 8 as follows:

$$ExporterTarget\_S4\_receiveMsg\_Trigger = t(d1, d2). \quad (34)$$

It is an example of receiving two messages in a single synchronization, which shows the importance of using polyadic π-calculus in the mapping.

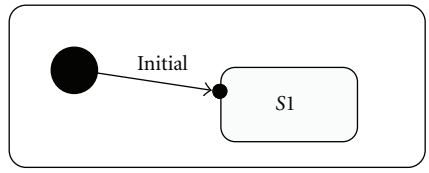


Figure 5: The state diagram of the capsule *TopSystem*.

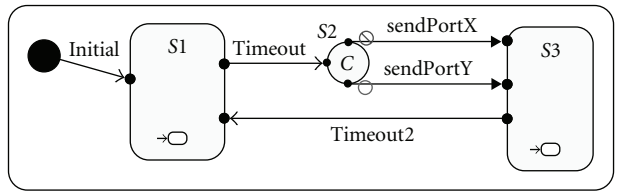


Figure 6: The state diagram of the capsule *Source*.

As an example to apply Definition 9, consider the transition segment *exportMsg*. So,

$$ExporterTarget\_S5\_exportMsg\_Action() = \overline{export}\langle d1 \rangle. \quad (35)$$

The next section presents three examples that illustrate how the mapping rules can be applied to obtain the π-calculus definitions from a UML-RT model, and how to manipulate the generated π-calculus definitions in order to verify some system requirements.

4. Examples of the Mapping

In this section, we develop three examples. The first one is the *TopSystem*, shown in Figure 1 and used in the previous sections to introduce the UML-RT and to exemplify the mapping rules application. The objective is to provide the entire UML-RT model of the *TopSystem* and to explain step by step how to obtain the π-calculus definition of its *Source* capsule. The example includes mobility and deals with some UML-RT concepts which were not previously addressed in our previous work, such as the entry and the exit actions, the composite states, and the transition chains.

The second example is a Heating System for Air Conditioning, which is an adaptation of [10], and the objective is to use the π-calculus definitions to verify if the UML-RT model correctly implements the system requirements. This example does not include mobility concept; however it is interesting to understand the possible manipulations using π-calculus reactions in order to verify the system behavior.

Finally, the third example is the Handover protocol, an example of mobile processes presented by Milner in [24]. It includes the communication of more than one message at a time, so it is suitable case for the polyadic π-calculus.

4.1. Router Example. In this section, we explain how to obtain the π-calculus definition of the entire capsule *TopSystem*, illustrated in Figure 1, using the mapping rules

Table 2: Entry and exit actions of the states in the capsule *TopSystem* and its subcapsules.

Capsule	State	Entry action	Exit action
TopSystem	*S*1	—	—
Source	*S*1	timer·informIn(RTTimespec(10, 0));	—
	*S*2	—	—
	*S*3	timer·informIn(RTTimespec(1, 0));	—
Router	*S*1	—	—
	*S*2	timer·informIn(RTTimespec(4, 0));	—
	*S*3	timer·informIn(RTTimespec(1, 0));	—
ConsumerTarget	*S*1	—	—
	*S*2	—	—
	*S*3	timer·informIn(RTTimespec(2, 0));	—
ExporterTarget	*S*1	—	—
	*S*2	—	—
	*S*3	timer·informIn(RTTimespec(1, 0));	—
	*S*4	—	—
	*S*5	—	—

provided in Section 3. The capsule *TopSystem* comprises the following capsules: *source* (instance of the capsule *Source*), *router* (instance of the capsule *Router*), *target1* (instance of the capsule *ConsumerTarget*), and *target2* (instance of the capsule *ExporterTarget*). We provide the state diagram of the capsules *TopSystem*, *Source*, *Router,* and *ConsumerTarget*, respectively, in Figures 5, 6, 7, and 8.

The entry and exit actions of the states in the capsule TopSystem and its subcapsules are detailed in Table 2. It is important to note that these actions are not considered as a sending message statement or a name reconfiguration statement, because they just set up an internal timeout, so the π-calculus definition of each action is the stop process.

The specification of the transitions inside the state diagrams of the capsule *TopSystem* and its subcapsules is provided in Table 3. Table 4 presented the pseudocode that represents the transition trigger and the transition action; the pseudocode helps to understand the application of the mapping rules to obtain the π-calculus definitions.

The structure diagrams of the *TopSystem* subcapsules are not provided because the majority of ports are public, so they can be seen in *TopSystem* structure diagram (shown in Figure 1). The only exception is the protected port *timer p*resented in all *TopSystem* subcapsules. The port *timer* implements the protocol *Timing* (a built-in protocol of the RoseRT tool) and receives the timeout signal.

In order to show how to apply the mapping rules to obtain the π-calculus definition of a UML-RT capsule, we detail the mapping of the capsule *Source*.

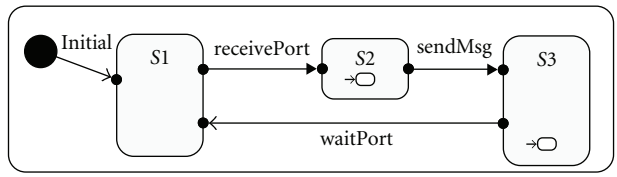


Figure 7: The state diagram of the capsule *Router*.

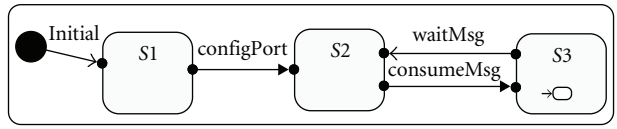


Figure 8: The state diagram of the capsule *ConsumerTarget*.

Using Definition 1, the π-calculus definition of the capsule *Source* is the parallel composition of its state diagram definition and its structure diagram definition:

$$Source(InputSet[Source]) = new(RestrictedSet[Source])\ (Source\_StructureDiagram \mid Source\_StateDiagram), \tag{36}$$

where

$$InputSet[Source] = \{q\}, \quad RestrictedSet[Source] = \{\} - \{q\} = \{\}. \tag{37}$$

So, we have

$$Source(q) = (Source\_StructureDiagram \mid Source\_StateDiagram). \tag{38}$$

According to Definition 2, the structure diagram of the capsule *Source* is the parallel composition of its subcapsules. As the capsule *Source* does not have subcapsules, the π-calculus definition of its structure diagram is the stop process:

$$Source\_StructureDiagram = 0. \tag{39}$$

The π-calculus definition of the *Source* state diagram, according to Definition 3, can be written as

$$Source\_StateDiagram = Source\_Initial\_Action \cdot Source\_S1\langle InputSet[Source, InitialPoint, S1]\rangle. \tag{40}$$

As the *Initial* transition does not have an action, using the Definition 9 it is omitted from the *Source* definition and we have

$$InputSet[Source, InitialPoint, S1] = \{q\}, \tag{41}$$

$$InputSet[Source, S1] = \{q\}, \tag{42}$$

$$Source\_StateDiagram = Source\_S1\langle q\rangle. \tag{43}$$

Table 3: UML-RT transitions' specification for the capsule *TopSystem* and its subcapsules.

Capsule	Transition	UML-RT trigger	UML-RT action
TopSystem	*Initial*	—	setCT·portName(x)·send(); setET·portName(y) ·send();
Source	*Initial*	—	—
	timeout	Port: *timer* Signal: *timeout*	—
	sendPortX	—	q·portName(x)·send();
	sendPortY	—	q·portName(y)·send();
	timeout2	—	—
Router	*Initial*	—	—
	receivePort	Port: *r* Signal: *portName*	a = *rtdata; alert·registerSPP(a);
	sendMsg	Port: *timer*	m·msg1 = $b1$; m·msg2 = $b2$;
		Signal: *timeout*	alert·msg(m)·send();
	waitPort	Port: *timer* Signal: *timeout*	—
ConsumerTarget	*Initial*	—	—
	configPort	Port: *setPort* Signal: *portName*	aux = *rtdata; s·registerSAP(aux);
	consumeMsg	Port: *s*	m = *rtdata; $c1$ = m·msg1;
		Signal: *msg*	$c2$ = m·msg2;
	waitMsg	Port: *timer* Signal: *timeout*	—
ExporterConsumer	*Initial*	—	—
	configPort	Port: *setPort* Signal: *portName*	aux = *rtdata; t·registerSAP(aux);
	Initial1	—	—
	Initial2	—	—
	receiveMsg	Port: *t*	m = *rtdata;
		Signal: *msg*	$d1$ = m·msg1; $d2$ = m·msg2;
	consumeMsg	—	—
	exportMsg	—	export·msg($d1$) ·send();
	msgConsumed	—	—
	msgExported	—	—
	waitMsg	Port: *timer* Signal: *timeout*	—

Using the Definition 4 and (42), the π-calculus definition of state $S1$, in the capsule *Source*, can be written as the sequence composition of its entry action and its unique outgoing transition

$$Source\_S1(q) = Source\_S1\_EntryAction \cdot Source\_S1\_timeout. \tag{44}$$

As the $S1$ entry action is not specified, according to Definition 5 it is omitted from the parent definition, so

$$Source\_S1(q) = Source\_S1\_timeout. \tag{45}$$

The π-calculus definition of the transition *timeout* is made based on Definition 6. So, from (45), we have

$$Source\_S1(q) = Source\_S1\_timeout\_trigger \cdot Source\_S1\_ExitAction \cdot P\_S1\_timeout\_Action \cdot Source\_S2\langle InputSet[Source, S1, S2]\rangle. \tag{46}$$

Table 4: Pseudocode of the transitions in the capsule *TopSystem* and its subcapsules.

Capsule	Transition	Trigger pseudocode	Action pseudocode
TopSystem	*Initial*	—	setCT send portName(x) setET send portName (y)
Source	*Initial*	—	—
	timeout	timer receive timeout( )	—
	sendPortX	—	q send portName(x)
	sendPortY	—	q send portName(y)
	timeout2	—	—
Router	*Initial*	—	—
	receivePort	r receive portName(a)	alert = a
	sendMsg	timer receive timeout( )	alert send msg($b1, b2$)
	waitPort	timer receive timeout( )	—
ConsumerTarget	*Initial*	—	—
	configPort	setPort receive portName (aux)	s = aux
	consumeMsg	s receive msg($c1, c2$)	—
	waitMsg	timer receive timeout( )	—
ExporterConsumer	*Initial*	—	—
	configPort	setPort receive portName(aux)	t = aux
	Initial1	—	—
	Initial2	—	—
	receiveMsg	t receive msg($d1, d2$)	—
	consumeMsg	—	—
	exportMsg	—	export send msg($d1$)
	msgConsumed	—	—
	msgExported	—	—
	waitMsg	timer receive timeout( )	—

The trigger of the transition *timeout*is the unobservable action τ, based on Definition 8, because the trigger in UML-RT is specified by the receiving of a timeout signal as stated in Table 4. The $S1$ exit action is not specified, so according to Definition 5 it is omitted from the parent definition. There is no action in transition *timeout*, so its definition is omitted from the parent definition and $InputSet[Source, S1, S2] = InputSet[Source, S2] = \{q\}$. From (46), the $S1$ definition in the π-calculus can be written as

$$Source\_S1(q) = \tau \cdot Source\_S2\langle q\rangle. \tag{47}$$

The $S1$ definition references the state $S2$. So, it is necessary to specify the π-calculus definition of state $S2$. So,*by*using Definition 4, we have

$$\begin{aligned} &Source\_S2(q) \\ &\quad = Source\_S2\_EntryAction \\ &\qquad \cdot (Source\_S2\_sendPortX + Source\_S2\_sendPortY). \end{aligned} \tag{48}$$

Using Definitions 5 and 6, from (48), we have

$$\begin{aligned} Source\_S2(q) = {} & ((Source\_S2\_sendPortX\_trigger \\ & \cdot Source\_S2\_ExitAction \\ & \cdot P\_S2\_sendPortX\_Action \\ & \cdot Source\_S3\langle InputSet[Source, S2, S3]\rangle) \\ & + (Source\_S2\_sendPortY\_trigger \\ & \cdot Source\_S2\_ExitAction \\ & \cdot P\_S2\_sendPortY\_Action \\ & \cdot Source\_S3\langle InputSet[Source, S2, S3]\rangle)). \end{aligned} \tag{49}$$

Using Definitions 5, 8, and 9, from (49) the π-calculus definition of $S2$ is written as

$$\begin{aligned} Source\_S2(q) = {} & (\overline{q}\langle x\rangle \cdot Source\_S3\langle q\rangle) \\ & + (\overline{q}\langle y\rangle \cdot Source\_S3\langle q\rangle). \end{aligned} \tag{50}$$

However, the $S2$ definition references the state $S3$. So, it is also necessary to specify the π-calculus definition of the state $S3$. So, by using Definition 4, we have

$$Source\_S3(q) = Source\_S3\_EntryAction \cdot Source\_S3\_timeout2. \tag{51}$$

Using Definitions 5 and 6, from (51), we have

$$\begin{aligned} Source\_S3(q) = {} & Source\_S3\_timeout2\_trigger \\ & \cdot Source\_S3\_ExitAction \\ & \cdot P\_S3\_timeout2\_Action \\ & \cdot Source\_S1\langle InputSet[Source, S3, S1]\rangle. \end{aligned} \tag{52}$$

Finally, using Definitions 5, 8, and 9, from (51) the π-calculus definition of $S3$ is written as

$$Source\_S3(q) = \tau \cdot Source\_S1\langle q\rangle. \tag{53}$$

From (47), (50), and (53), the π-calculus definition of the state $S1$ is

$$\begin{aligned} Source\_S1(q) = {} & \tau \cdot ((\overline{q}\langle x\rangle \cdot \tau \cdot Source\_S1\langle q\rangle) \\ & + (\overline{q}\langle y\rangle \cdot \tau \cdot Source\_S1\langle q\rangle)). \end{aligned} \tag{54}$$

Finally, using (38), (39), (43), and (54), the π-calculus definition of the capsule *Source* is given by

$$\begin{aligned} Source(q) = {} & Source\_S1\langle q\rangle \\ Source\_S1(q) = {} & \tau \cdot ((\overline{q}\langle x\rangle \cdot \tau \cdot Source\_S1\langle q\rangle) \\ & + (\overline{q}\langle y\rangle \cdot \tau \cdot Source\_S1\langle q\rangle)). \end{aligned} \tag{55}$$

Using the mapping rules and the specifications of transitions in Table 4, it is possible to define the π-calculus definition of the other capsules that compose the *TopSystem* example, as written in

$$\begin{aligned} Router(r) = {} & Router\_S1\langle r\rangle \\ Router\_S1(r) = {} & r(a) \cdot Router\_S2\langle a, r\rangle \\ Router\_S2(alert, r) = {} & \tau \cdot \overline{alert}\langle b1, b2\rangle \cdot \tau \cdot Router\_S1\langle r\rangle \end{aligned} \tag{56}$$

$$\begin{aligned} & ConsumerTarget(setPort) \\ & \quad = setPort(aux) \\ & \qquad \cdot ConsumerTarget\_S2\langle aux, setPort\rangle \\ & ConsumerTarget\_S2(s, setPort) \\ & \quad = s(c1, c2) \cdot \tau \\ & \qquad \cdot ConsumerTarget\_S2\langle s, setPort\rangle \end{aligned} \tag{57}$$

$$\begin{aligned} & ExporterTarget(setPort, export) \\ & \quad = setPort(aux) \\ & \qquad \cdot ExporterTarget\_S4\langle aux, setPort, export\rangle \\ & ExporterTarget\_S4(t, setPort, export) \\ & \quad = t(d1, d2) \\ & \qquad \cdot (\tau \cdot ExporterTarget\_S4\langle t, setPort, export\rangle \\ & \qquad\quad + (\overline{export}\langle d1\rangle \cdot \tau \\ & \qquad\qquad \cdot ExporterTarget\_S4\langle t, setPort, export\rangle)) \end{aligned} \tag{58}$$

$$\begin{aligned} & TopSystem(out) \\ & = new\ z, setCT, setET \\ & \quad \cdot (Source\langle z\rangle \mid Router\langle z\rangle \mid ConsumerTarget\langle setCT\rangle \\ & \quad \cdot \mid ExporterTarget\langle setET, out\rangle \mid \overline{setCT}\langle x\rangle \cdot \overline{setET}\langle y\rangle\Big). \end{aligned} \tag{59}$$

Through the definition of a capsule it is possible to capture the behavior of this capsule, for example, the capsule *Source* in (55) executes an unobservable action (i.e., a timeout), sends the message x or y along the port q, and returns to a state (in this case *S1*), where it is possible to repeat the same behavior. Due to the connection between the subcapsules *Source* and *Router* in the capsule *TopSystem*, the parent capsule references its subcapsule *Source* renaming the port q to z, and references its subcapsule *Router* renaming the port r to z. Therefore, the message x or y sent by the subcapsule *Source* is received by the subcapsule *Router*, and it is stored in the name a, according to (56). The definition of *Router_S1* references the definition of *Router_S2* renaming *alert* with a, so the subcapsule *Router* is now able to send the messages $b1$ and $b2$ through the port *alert*, which can be x or y.

The capsule *TopSystem* in (59) sends x to its subcapsule *ConsumerTarget* and sends y to its subcapsule *ExporterTarget*.

According to (57), the subcapsule *ConsumerTarget* configures its port s with x to be able to receive the message from the capsule *Router*. Similarly, according to (58), the subcapsule *ExporterTarget* configures its port t with y to be able to receive the message from the capsule *Router*. After receiving the message from the *Router*, the control of the subcapsule *ExporterTarget* can return to *ExporterTarget_S*4, or it can send the received message through its port *export* and later the control returns to *ExporterTarget_S*4. Note that the port *export* is in fact renamed to *out*, which is a public interface of the capsule *TopSystem*, according to (59).

In the example, the π-calculus definitions corresponding to the UML-RT model were specified. Using the generated π-calculus definitions, it is possible to analyze the interactions between the system components, therefore some undesirable system behavior can be uncovered during the modeling phase.

4.2. Heating System. In this section, we use an example of a system, which was adapted from the example described in [10] and is called Heating System for Air Conditioning. In this example, the system requirements are provided as well as the UML-RT model that implements the requirements. Furthermore, the π-calculus definitions for the system are generated following the mapping rules. Some scenarios are proposed according to the system requirements. Later, we compose the π-calculus definitions of the system and the scenarios in order to verify if the model meets the requirements. The handlings are made manually.

The heating system consists of the standard heating subsystem and the additional heating subsystem that is used to shorten the time needed to increase the temperature in the car. The specification of the heating system is provided through the requirements R1 to R7. In the requirements' descriptions we anticipate the UML-RT states of the state diagrams of this section. They are named between parentheses. The requirements are as follows.

R1. The heating system has two levels: level1 (*Level1* state) and level2 (*Level2* state). In level1 only the standard heating system is active, while in level2 the additional heating system is also active.

R2. An incoming init event changes the heating system's status from shutdown (*Shutdown* state) to level1 and from start (*Start* state) to level2.

R3. The heating system automatically changes its status from level2 to level1 after 10 minutes.

R4. The heating system automatically changes its status from start to shutdown after 10 minutes.

R5. Exceptions regarding invalid voltage or ignition key in status "cold" immediately cause deactivation of the heating system for 10 seconds.

R6. As soon as there is no more exception, the heating system continues operating on the selected level.

R7. The heating system' status changes to shutdown if the ignition key stays in status "cold" for 5 minutes, or if the key is removed.

The heating system model (Figure 9) was constructed using the RoseRT tool. It is a capsule called *HeatingSystem* composed of three subcapsules: *low* (instance of the capsule *LowExceptions* in Figure 10), *high* (instance of the capsule *HighExceptions* in Figure 11), and *controller* (instance of the capsule *HeatingController* in Figure 12). The interface of the *HeatingSystem* is the ports to receive the ignition status (*KL15off*, *KL15radio,* and *KL15cold*), the voltage invalid status (*iVolt*), the signal to start and shutdown the controller operation (*startController* and *shutdownController*, resp.). The specification of the transitions inside each capsule state diagram is in Table 5; whereas the entry and exit actions of the states are detailed in Table 6.

The capsule *LowExceptions* starts at the state *OkayLPrio* and changes to *OffLPrio*, if it receives a signal through the port *ignitionCold* or *invalidVoltage*. During this transition, the capsule sends a signal through the port *off* in order to shutdown the heating controller temporarily. After 10 seconds, the capsule *LowExceptions* is ready to receive more exception information, when it returns to the state *OkayLPrio*.

The capsule *HighExceptions* starts at the state *OffHPrio* and changes to *OkayHPrio*, if it receives a signal through the port *ignitionRadio*. This transition executes an action of sending a signal along the port *on* in order to start the heating controller, because there is no more high exception to be handled. After receiving a signal through the port *ignitionCold* at the state *OkayHPrio*, the capsule *HighExceptions* changes to the state *WaitHPrio*. At the state *WaitHPrio*, the capsule can change to the state *OkayHPrio* if the ignition returns to the status radio; or it can detect a high exception if the ignition exceeds 5 minutes in the status cold, or if the ignition changes to the status off. In case of detecting a high exception, the capsule sends a signal along the port *off* aiming to shutdown the heating controller.

The capsule *HeatingController* begins at the state *Start* and changes to *Level2* if it receives a signal through the ports *on* or *init*. At the state *Level2*, the capsule can changes to the state *Level1* after 10 minutes, or it can shutdown after receiving a signal along the ports *off* or *end*. At the state *Level1*, the capsule can shutdown too. The capsule is able to notify its status in each state sending a message through the port *out*.

Some system requirements can be verified by a visual inspection in the model, for example, the requirements R1 to R4 are addressed by the transitions specified in the state diagram of the capsule *HeatingController*. Other requirements are modeled, for example, R5 to R7, however we do not know if the behavior of the system interacting components complies with the requirements. In order to verify some properties originated immediately from the requirements, we can use the system formal specification in the π-calculus.

Table 5: Transitions in the *HeatingSystem* subcapsules.

Capsule	Transition	UML-RT trigger	UML-RT action
LowExceptions	*Initial*	—	—
	receiveException	Port: *ignitionCold* Signal: *sig* Port: *invalidVoltage* Signal: *sig*	off·sig()·send();
	expireException	Port: *timer* Signal: *timeout*	on·sig()·send();
HighExceptions	*Initial*	—	—
	receiveIgnitionKeyRadio	Port: *ignitionRadio* Signal: *sig*	on·sig()·send();
	receiveIgnitionKeyCold	Port: *ignitionCold* Signal: *sig*	—
	receiveIgnitionKeyRadio2	Port: *ignitionRadio* Signal: *sig*	—
	receiveIgnitionKeyOff	Port: *ignitionOff* Signal: *sig*	off·sig()·send();
	after 5 *minutes*	Port: *timer* Signal: *timeout*	off·sig()·send();
HeatingController	*Initial*	—	—
	Initiate	Port: *on* Signal: *sig* Port: *init* Signal: *sig*	—
	Terminate	Port: *off* Signal: *sig* Port: *end* Signal: *sig*	—
	After 10 *minutes*	Port: *timer* Signal: *timeout*	—

Table 6: Entry and exit actions of the states in the *HeatingSystem* subcapsules.

Capsule	State	Entry action	Exit action
LowExceptions	*OkayLPrio*	—	—
	OffLPrio	timer·informIn(RTTimespec(10, 0));	—
HighExceptions	*OffHPrio*	—	—
	OkayHPrio	—	—
	WaitHPrio	timer·informIn(RTTimespec(300, 0));	—
HeatingController	*Start*	timer·informIn(RTTimespec(600, 0));	—
	Level2	out·msg(L2)·send(); timer·informIn(RTTimespec(600, 0));	—
	Level1	out·msg(L1) ·send();	—
	Shutdown	out·msg(BAD) ·send();	—

The π-calculus definitions of the heating system are provided below; however we omit the capsule name in each state name due to space limitation

$$\begin{aligned} &HeatingSystem(KL15radio, KL15cold,\\ &\qquad KL15\text{off}, iVolt, startController,\\ &\qquad shutdownController, heatingStatus)\\ &\quad = new\ on, \text{off}\,(HeatingController\langle Set1\text{Ref}\rangle\\ &\qquad \mid LowExceptions\langle Set2\text{Ref}\rangle\\ &\qquad \mid HighExceptions\langle Set3\text{Ref}\rangle), \end{aligned} \tag{60}$$

where

$$\begin{aligned} Set1\text{Ref} &= \{startController, shutdownController,\\ &\quad on, \text{off}, heatingStatus\},\\ Set2\text{Ref} &= \{KL15cold, iVolt, on, \text{off}\},\\ Set3\text{Ref} &= \{KL15radio, KL15cold, KL15\text{off}, on, \text{off}\}. \end{aligned} \tag{61}$$

Declaring $Set1 = \{init, end, on, \text{off}, out\}$ as the input set of *HeatingController* definition, we have

$$\begin{aligned} &HeatingController(List1) = Start\langle Set1\rangle\\ &Start(Set1)\\ &\quad = (on + init)\cdot Level2\langle Set1\rangle + \tau\cdot Shutdown\langle Set1\rangle\\ &Level2(Set1)\\ &\quad = \overline{out}\langle L2\rangle\cdot((\text{off} + end)\cdot Shutdown\langle Set1\rangle\\ &\qquad +\tau\cdot Level1\langle Set1\rangle)\\ &Level1(Set1)\\ &\quad = \overline{out}\langle L1\rangle\cdot((\text{off} + end)\cdot Shutdown\langle Set1\rangle)\\ &Shutdown(Set1)\\ &\quad = \overline{out}\langle BAD\rangle\cdot((on + init)\cdot Level1\langle Set1\rangle). \end{aligned} \tag{62}$$

Declaring $et2 = \{ignitionCold, invalidVoltage, on, \text{off}\}$ as the input set of *LowExceptions* definition, we have

$$\begin{aligned} &LowExceptions(Set2) = OkayLPrio\langle Set2\rangle\\ &OkayLPrio(Set2) = (ignitionCold + invalidVoltage)\\ &\qquad \cdot \overline{\text{off}}\cdot \text{Off}LPrio\langle Set2\rangle\\ &\text{Off}LPrio(Set2) = \tau\cdot\overline{on}\cdot OkayLPrio\langle Set2\rangle. \end{aligned} \tag{63}$$

Declaring $Set3 = \{ignitionRadio, ignitionCold, ignitionOff, on, \text{off}\}$ as the input set of *HighExceptions* definition, we have

$$\begin{aligned} &HighExceptions(Set3)\\ &\quad = \text{Off}HPrio\langle Set3\rangle\\ &\text{Off}HPrio(Set3)\\ &\quad = ignitionRadio\cdot\overline{on}\cdot OkayHPrio\langle Set3\rangle\\ &OkayHPrio(Set3)\\ &\quad = ignitionCold\cdot WaitHPrio\langle Set3\rangle\\ &WaitHPrio(Set3)\\ &\quad = ignitionRadio\cdot OkayHPrio\langle Set3\rangle + ignition\text{Off}\\ &\qquad \cdot\overline{\text{off}}\cdot \text{Off}HPrio\langle Set3\rangle + \tau\cdot\overline{\text{off}}\\ &\qquad \cdot \text{Off}HPrio\langle Set3\rangle. \end{aligned} \tag{64}$$

After obtaining the definitions of the components in the model, it is desirable to verify its properties. In order to verify the requirements R5 and R6, we assume the scenario where an external actor sends a $\overline{startController}$ event and, later, it sends an $\overline{iVolt}$ event. The scenario can be specified as

$$\begin{aligned} Scenario1 = {}&\overline{startController}\cdot heatingStatus(s1)\\ &\cdot\overline{iVolt}\cdot heatingStatus(s2)\cdot heatingStatus(s3). \end{aligned} \tag{65}$$

After the $\overline{startController}$ event, the information that the system is at the level2 is expected, so $s1$ will receive "$L2$". After the $\overline{iVolt}$ event, the information that the system is shutdown is due, so $s2$ will receive "BAD". After the timeout of the low exception (in this case, the invalid voltage), the information that the system is at the level2 is waited, so $s3$ will be "$L2$", because the requirement R6 specifies that the system has to return to the selected level after the exceptions.

Composing the heating system and the scenario, we obtain

$$\begin{aligned} &HeatingSystem(KL15radio, KL15cold, KL15\text{off}, iVolt,\\ &\qquad startController, shutdowController,\\ &\qquad heatingStatus)\mid Scenario1. \end{aligned} \tag{66}$$

Using (60) and (65), we have

$$\begin{aligned} &new\ on, \text{off}\,(HeatingController\langle Set1\text{Ref}\rangle\\ &\qquad \mid LowExceptions\langle Set2\text{Ref}\rangle\\ &\qquad \mid HighExceptions\langle Set3\text{Ref}\rangle)\\ &\qquad \mid (\overline{startController}\cdot heatingStatus(s1)\\ &\qquad\ \cdot\overline{iVolt}\cdot heatingStatus(s2)\\ &\qquad\ \cdot heatingStatus(s3)). \end{aligned} \tag{67}$$

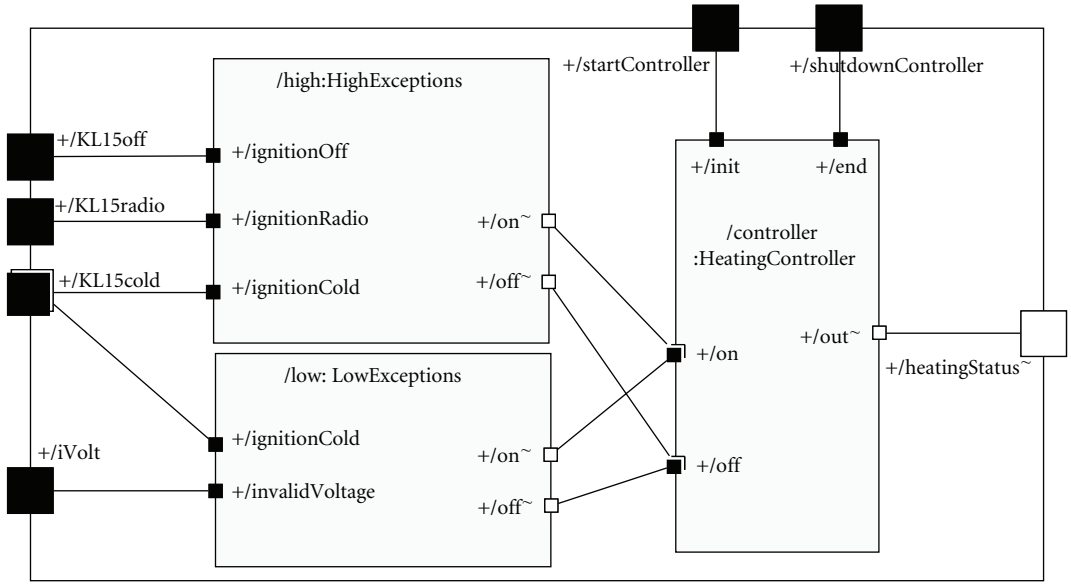


Figure 9: The structure diagram of the capsule *HeatingSystem*.

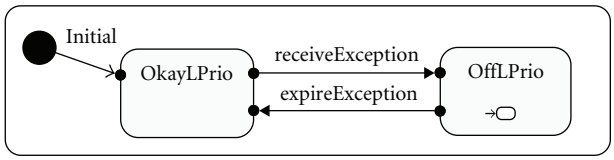


Figure 10: The state diagram of the capsule *LowExceptions*.

Replacing the definitions of the *HeatingSystem* subcapsules, we obtain

$$\begin{aligned}
&\textit{new on, off}(((on + startController) \cdot Level2\langle Set1\text{Ref}\rangle \\
&\qquad +\tau \cdot Shutdown\langle Set1\text{Ref}\rangle) \\
&\quad \mid \big((KL15cold + iVolt) \cdot \overline{\text{off}} \\
&\quad \cdot \text{Off}LPrio\langle Set2\text{Ref}\rangle) \\
&\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\text{Ref}\rangle) \\
&\quad \mid \big(\overline{startController} \cdot heatingStatus(s1) \\
&\qquad \cdot \overline{iVolt} \cdot heatingStatus(s2) \\
&\qquad \cdot heatingStatus(s3))\big).
\end{aligned} \tag{68}$$

After the reaction between $\overline{startController}$ and *startController*, we have

$$\begin{aligned}
&\textit{new on, off}(Level2\langle Set1\text{Ref}\rangle \\
&\quad \mid \big((KL15cold + iVolt) \cdot \overline{\text{off}} \\
&\quad \cdot \text{Off}LPrio\langle Set2\text{Ref}\rangle) \\
&\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\text{Ref}\rangle) \\
&\quad \mid \big(heatingStatus(s1) \cdot \overline{iVolt} \\
&\qquad \cdot heatingStatus(s2) \\
&\qquad \cdot heatingStatus(s3))\big).
\end{aligned} \tag{69}$$

Replacing the definition of $Level2\langle Set1\text{Ref}\rangle$, we obtain

$$\begin{aligned}
&\textit{new on, off}\big(\big(\overline{heatingStatus}\langle L2\rangle \\
&\qquad \cdot ((\text{off} + shutdownController) \\
&\qquad \cdot Shutdown\langle Set1\text{Ref}\rangle \\
&\qquad +\tau \cdot Level1\langle Set1\text{Ref}\rangle)) \\
&\quad \mid \big((KL15cold + iVolt) \cdot \overline{\text{off}} \\
&\qquad \cdot \text{Off}LPrio\langle Set2\text{Ref}\rangle) \\
&\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\text{Ref}\rangle) \\
&\quad \mid \big(heatingStatus(s1) \cdot \overline{iVolt} \\
&\qquad \cdot heatingStatus(s2) \\
&\qquad \cdot heatingStatus(s3))\big).
\end{aligned} \tag{70}$$

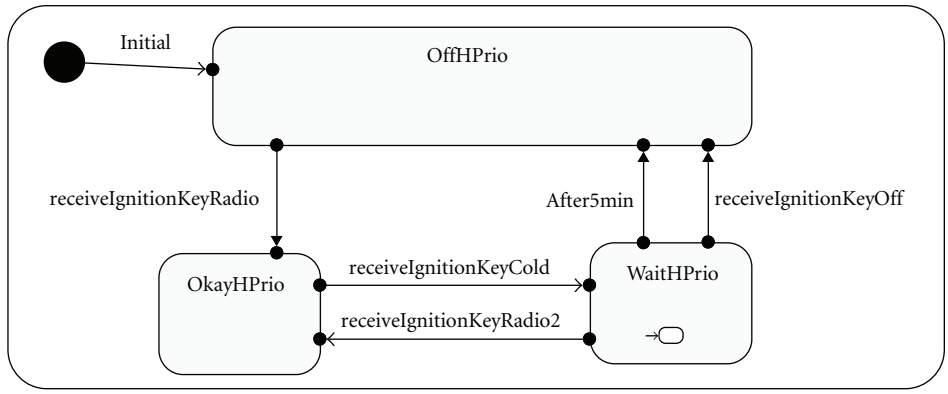


Figure 11: The state diagram of the capsule *HighExceptions*.

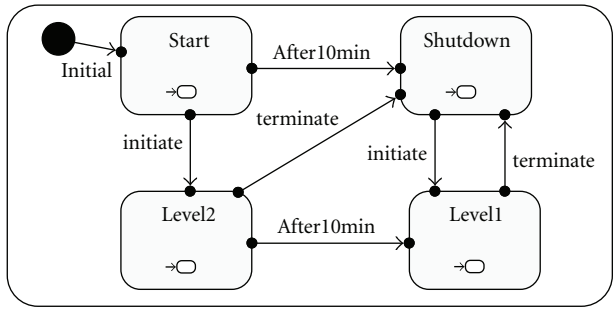


Figure 12: The state diagram of the capsule *HeatingController*.

After the reaction between $\overline{heatingStatus}\langle L2\rangle$ and $heatingStatus(s1)$, the name $s1$ receives the message "$L2$" as required in the scenario specification. So,

$$\begin{aligned} &new\ on, \mathrm{off}(((\mathrm{off} + shutdownController) \\ &\quad \cdot Shutdown\langle Set1\mathrm{Ref}\rangle \\ &\quad + \tau \cdot Level1\langle Set1\mathrm{Ref}\rangle) \\ &\quad \mid \left((KL15cold + iVolt) \cdot \overline{\mathrm{off}}\right. \\ &\quad \left. \cdot \mathrm{Off}LPrio\langle Set2\mathrm{Ref}\rangle\right) \\ &\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\mathrm{Ref}\rangle) \\ &\quad \mid \left(\overline{iVolt} \cdot heatingStatus(s2)\right. \\ &\quad \left. \cdot heatingStatus(s3)\right)). \end{aligned} \tag{71}$$

After the reaction between $\overline{iVolt}$ and $iVolt$, we have

$$\begin{aligned} &new\ on, \mathrm{off}(((\mathrm{off} + shutdownController) \\ &\quad \cdot Shutdown\langle Set1\mathrm{Ref}\rangle \\ &\quad + \tau \cdot Level1\langle Set1\mathrm{Ref}\rangle) \\ &\quad \mid \left(\overline{\mathrm{off}} \cdot \mathrm{Off}LPrio\langle Set2\mathrm{Ref}\rangle\right) \\ &\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\mathrm{Ref}\rangle) \\ &\quad \mid (heatingStatus(s2) \cdot heatingStatus(s3))). \end{aligned} \tag{72}$$

After the reaction between $\overline{\mathrm{off}}$ and off, we obtain

$$\begin{aligned} &new\ on, \mathrm{off}(Shutdown\langle Set1\mathrm{Ref}\rangle \mid \mathrm{Off}LPrio\langle Set2\mathrm{Ref}\rangle \\ &\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\mathrm{Ref}\rangle) \\ &\quad \mid (heatingStatus(s2) \cdot heatingStatus(s3))). \end{aligned} \tag{73}$$

Replacing the definition of *Shutdown* and *OffLPrio*, we have

$$\begin{aligned} &new\ on, \mathrm{off}\Big(\left(\overline{heatingStatus}\langle BAD\rangle\right. \\ &\quad \left. \cdot((on + startController) \cdot Level1\langle Set1\mathrm{Ref}\rangle)\right) \\ &\quad \mid (\tau \cdot \overline{on} \cdot OkayLPrio\langle Set2\mathrm{Ref}\rangle) \\ &\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\mathrm{Ref}\rangle) \\ &\quad \mid (heatingStatus(s2) \cdot heatingStatus(s3))). \end{aligned} \tag{74}$$

After the reaction between $\overline{heatingStatus}\langle BAD\rangle$ and $heatingStatus(s2)$, the name $s2$ receives the message "BAD" as required in the scenario specification. So,

$$\begin{aligned} &new\ on, \mathrm{off}(((on + startController) \cdot Level1\langle Set1\mathrm{Ref}\rangle) \\ &\quad \mid (\tau \cdot \overline{on} \cdot OkayLPrio\langle Set2\mathrm{Ref}\rangle) \\ &\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\mathrm{Ref}\rangle) \\ &\quad \mid heatingStatus(s3)). \end{aligned} \tag{75}$$

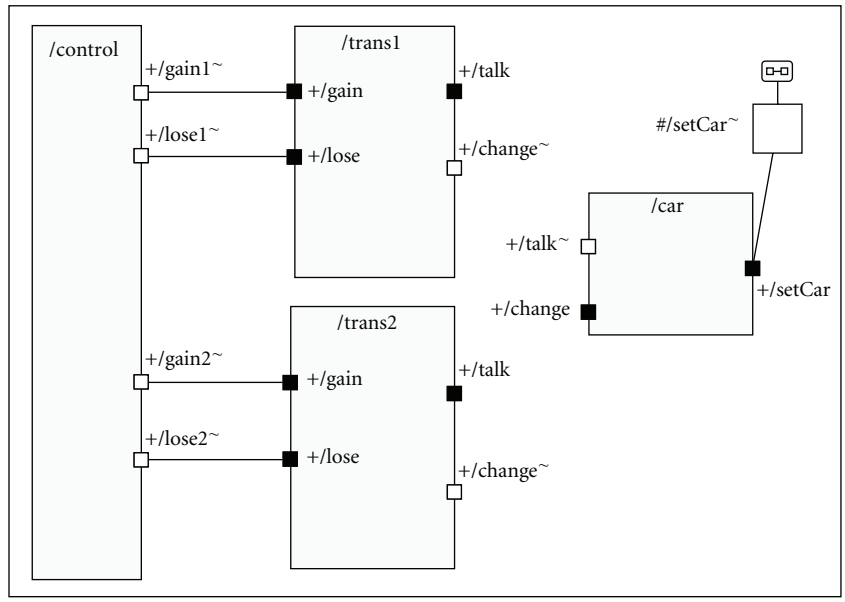


Figure 13: The structure diagram of the capsule *Handover*.

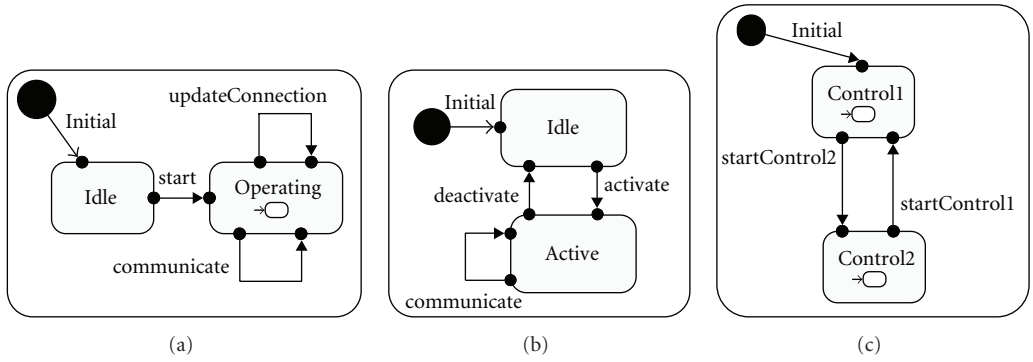


Figure 14: The state diagrams of the capsules: (a) *Car,* (b) *Trans, and* (c) *Control.*

After the τ reaction that represents the 10 seconds after the low exception, we obtain

$$\begin{aligned} &new\ on, \text{off}(((on + startController) \cdot Level1\langle Set1\text{Ref}\rangle) \\ &\quad \mid (\overline{on} \cdot OkayLPrio\langle Set2\text{Ref}\rangle) \\ &\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\text{Ref}\rangle) \\ &\quad \mid heatingStatus(s3)). \end{aligned} \tag{76}$$

After the reaction between $\overline{on}$ and on, we have

$$\begin{aligned} &new\ on, \text{off}(Level1\langle Set1\text{Ref}\rangle \mid OkayLPrio\langle Set2\text{Ref}\rangle \\ &\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\text{Ref}\rangle) \\ &\quad \mid heatingStatus(s3)). \end{aligned} \tag{77}$$

Replacing the definition of *OkayLPrio*, we obtain

$$\begin{aligned} &new\ on, \text{off}\Big(\Big(\overline{heatingStatus}\langle L1\rangle \\ &\quad \cdot (\text{off} + shutdownController) \\ &\quad \cdot shutdown\langle Set1\text{Ref}\rangle)) \\ &\quad \mid OkayLPrio\langle Set2\text{Ref}\rangle \\ &\quad \mid (KL15radio \cdot \overline{on} \cdot OkayHPrio\langle Set3\text{Ref}\rangle) \\ &\quad \mid heatingStatus(s3)). \end{aligned} \tag{78}$$

Now there is the possibility of the reaction between $\overline{heatingStatus}\langle L1\rangle$ and $heatingStatus(s3)$, then the name $s3$ receives the message "$L1$". Note that the heating system is in level1, which is an active status, so the requirement R5 is satisfied. Otherwise, the requirement R6 is not met, because the heating system was at level2 before the low exception and

Table 7: Transitions in the capsule *Handover* and its subcapsules.

Capsule	Transition	UML-RT trigger	UML-RT action
Handover	*Initial*	—	$m\cdot$talk = talk1; $m\cdot$change = change1; setCar$\cdot$signal(m)$\cdot$send();
Car	*Initial*	—	—
	start	Port: *setCar*	m = *rtdata; $t = m\cdot$talk; $c = m\cdot$change;
		Signal: *signal*	talk$\cdot$registerSAP(t); change$\cdot$registerSAP(c);
	updateConnection	Port: *change*	m = *rtdata; $t = m\cdot$talk; $c = m\cdot$change;
		Signal: *signal*	talk$\cdot$registerSAP(t); change$\cdot$registerSAP(c);
	communicate	Port: *timer* Signal: *timeout*	talk$\cdot$signal(info)$\cdot$send();
Trans	*Initial*	—	—
	activate	Port: *gain*	m = *rtdata; $t = m\cdot$talk; $c = m\cdot$change;
		Signal: *signal*	talk$\cdot$registerSPP(t); change$\cdot$registerSPP(c);
	deactivate	Port: *lose*	m = *rtdata; $t = m\cdot$talk; $c = m\cdot$change
		Signal: *signal*	$m2\cdot$talk = t; $m2\cdot$change = c; change$\cdot$signal($m2$)$\cdot$send();
	communicate	Port: *talk* Signal: *signal*	myinfo = *rtdata;
Control	*Initial*	—	$m\cdot$talk = talk1; $m\cdot$change = change1; gain1$\cdot$signal(m)$\cdot$send();
	startControl1	Port: *timer*	$m\cdot$talk = talk1; $m\cdot$change = change1;
		Signal: *timeout*	lose2$\cdot$signal(m)$\cdot$send(); gain1$\cdot$signal(m)$\cdot$send();
	startControl2	Port: *timer*	$m\cdot$talk = talk2; $m\cdot$change = change2;
		Signal: *timeout*	lose1$\cdot$signal(m)$\cdot$send(); gain2$\cdot$signal(m)$\cdot$send();

now it is at level1, when it should continue in level2. In this example, the designer can uncover the error in the model analyzing the state diagrams; however in a more complex system, the visual analysis is very difficult, so the formal analysis is necessary.

Aiming to verify the requirement R7, we propose a scenario composed by the following steps: the ignition key is in status radio, an external event initiates the controller and, later, the ignition key is in status off. The scenario is specified as

$$\begin{aligned} Scenario2 = {} & \overline{KL15radio} \cdot \overline{startController} \\ & \cdot heatingStatus(s1) \\ & \cdot \overline{KL15\text{off}} \cdot heatingStatus(s2). \end{aligned} \tag{79}$$

After the $\overline{startController}$ event, it is expected to be informed that the system is at the level2, so $s1$ will receive "$L2$". After the $\overline{KL15off}$ event, it is expected to be informed that the system is shutdown, so $s2$ will receive "BAD". Analyzing this scenario, we note that, after receiving the event $\overline{startController}$, the *HighExceptions* capsule sends the event $\overline{on}$ to initiate the heating controller. So, a component to manage exceptions can initiate the controller for the first time, before it has been initiated by the external event $\overline{startController}$. In this case, we obtain an unexpected behavior according to the proposed scenario; however we are not sure if it is an error in the model or a missing requirement specification.

Then, using the system specification in the π-calculus and the scenarios or properties proposed according to the system requirements, it is possible to analyze the interaction between the system components and find out the undesirable behavior due to errors in modeling or and a lack of specification.

4.3. Handover Protocol. In this section, we show the example of mobile processes called Handover Protocol presented by Milner in [24]. The *Handover* system (Figure 13) is composed by a control tower (*Control* capsule), two communicating towers (*Trans* capsules: *trans1* and *trans2*), and a car (capsule *Car*). For a better visualization of Figure 13, the protocol type of each port is omitted. The control tower has fixed connections with the communicating towers; it is expressed through the wired ports between the capsule *Control* and the two capsules *Trans*. The car communicates with a *Trans* tower, but an event can make it communicates with the other *Trans*.

The state diagrams of the capsules *Car*, *Trans,* and *Control* are presented in Figure 14. The state diagram of the top level capsule *Handover* is not shown; however it is composed only by the initial transition and an initial state called *S*1. The UML-RT specifications for the transitions of all capsules are available in Table 7.

The overall behavior is the following. the transition *Initial* of the capsule *Handover* sends two messages *talk1* and *change1* along the protected *setCar*. The messages are received by the *car* in order to make the initial configuration of its unwired ports *talk* and *change*. On the other hand, when *control* initializes, it sends to *trans1* the messages *talk1* and *change1* that are used by *trans1* to configure the unwired ports *talk* and *change*. At this moment, the *car* is able to communicate to *trans1* by sending a message through the port *talk*.

However, *control*can activate the *trans2*. In this case, the *control* sends the messages *talk2* and *change2* to *trans1* (via the port *lose1*) and to *trans2* (via the port *gain2*). The *trans1* receives the messages *talk2* and *change2*, and it sends to the *car* (via the port *change*). So, the *car* uses the messages to reconfigure its unwired ports *talk* and *change*. Besides, the *trans2* receives the messages *talk2* and *change2* from the *control* to configure its unwired ports *talk* and *change*. At this moment, the *car* is able to communicate to *trans2* by sending a message through the port *talk*.

The *control* continues its processing either activating *trans1* or activating *trans2*. It allows the *car* always to communicate to some tower, that is, *trans1* or *trans2*.

It is desirable to obtain the π-calculus definition of the Handover system, so it is important to analyze each capsule based on Figure 14 and Table 7in order to apply the mapping rules proposed in Section 3.

Analyzing the capsule *Car*, in the transition *start*, it receives the messages t and c in order to make the initial configuration of its unwired ports. After the initial configuration, *Car* in the state *Operating* is ready to send the message *info* to a tower through the port *talk*. However, *Car* can also receive from *trans1* two messages (t and c) simultaneously along the port *change*, indicating a new name for the port *talk* and a new name for the port *change*, respectively. This is the step that the car reconfigures its ports to start to operate with the other tower. Then, the π-calculus definition of the capsule *Car* is written as

$$\begin{aligned}
&Car(setCar)\\
&\quad = Car_Idle\langle setCar\rangle\\
&Car_Idle(setCar)\\
&\quad = setCar(t,c,setCar)\cdot Car_Operating\langle t,c,setCar\rangle\\
&Car_Operating(talk,change,setCar)\\
&\quad = \tau\cdot\overline{talk}\langle info\rangle\cdot Car_Operating(talk,change,setCar)\\
&\qquad + change(t,c)\cdot Car_Operating(talk,change,setCar).
\end{aligned} \tag{80}$$

The tower *Trans* begins in the state *Idle,* where it is possible to receive the configuration for its unwired ports *talk* and *change* via the transition *activate*. In the *Active* state, *Trans* can receive a message *myinfo* from *Car* using the transition *communicate*. However, *Trans* can also receive a message from *Control* with the new names t and c. These names are later sent to *Car* via the port *change* and indicate how its unwired ports must be reconfigured. Using the mapping rules, the capsule *Trans* is defined in the π-calculus as

$$\begin{aligned}
&Trans(gain,lose)\\
&\quad = Trans_Idle\langle gain,lose\rangle\\
&Trans_Idle(gain,lose)\\
&\quad = gain(t,c)\cdot Trans_Active(t,c,gain,lose)\\
&Trans_Active(talk,change,gain,lose)\\
&\quad = talk(myinfo)\cdot Trans_Active(talk,change,gain,lose)\\
&\qquad + lose(t,c)\cdot\overline{change}\langle t,c\rangle\cdot Trans_Idle\langle gain,lose\rangle.
\end{aligned} \tag{81}$$

The state diagram of *Control* initiates (transition *Initial*) sending the messages *talk1* and *change1* along the port *gain1*. In the *Control1* state, after a timeout (transition

startControl2), *Control* can send the messages *talk2* and *change2* through the port *lose1*, and later through the port *gain2*. In a similar way, in the *Control2* state, after a timeout (transition *startControl1*), *Control* can send the messages *talk1* and *change1* through the port *lose2*, and later through the port *gain1*. In this way, the transition *startControl2* indicates that *trans2* is apt to talk with the car; while the transition *startControl1* indicates that *trans1* is now ready to talk with the car. Then, the π-calculus definition of the capsule *Control* is written as

$$\begin{aligned}
&Control(gain1, lose1, gain2, lose2)\\
&\quad = \overline{gain1}\langle talk1, change1\rangle\\
&\qquad \cdot Control_Control1\langle gain1, lose1, gain2, lose2\rangle\\
&Control_Control1(gain1, lose1, gain2, lose2)\\
&\quad = \tau \cdot \overline{lose1}\langle talk2, change2\rangle \cdot \overline{gain2}\langle talk2, change2\rangle\\
&\qquad \cdot Control_Control2\langle gain1, lose1, gain2, lose2\rangle\\
&Control_Control2(gain1, lose1, gain2, lose2)\\
&\quad = \tau \cdot \overline{lose2}\langle talk1, change1\rangle \cdot \overline{gain1}\langle talk1, change1\rangle\\
&\qquad \cdot Control_Control2\langle gain1, lose1, gain2, lose2\rangle.
\end{aligned} \tag{82}$$

Finally, the *Handover* system is defined in the π-calculus as the parallel composition of the definition of its state diagram and its structure diagram definition. The state diagram definition is the possibility to send the messages *talk1* and *change1* along the port *setCar*. The structure diagram definition includes a reference to each subcapsule definition. So, the *Handover*definition is written as

$$\begin{aligned}
&Handover()\\
&\quad = new(setCar, zgain1, zlose1, zgain2, zlose2)\\
&(Car\langle setCar\rangle \mid Trans\langle zgain1, zlose1\rangle\\
&\quad \mid Trans\langle zgain2, zlose2\rangle\\
&\quad \mid Control\langle zgain1, zlose1, zgain2, zlose2\rangle\\
&\quad \mid \overline{setCar}\langle talk1, change1\rangle\Big).
\end{aligned} \tag{83}$$

It is important to understand each term in the parallel composition shown in (83). The first term represents the subcapsule *car* (instance of *Car*). The second and the third terms represent the subcapsules *trans1* and *trans2* (instance of *Trans*), respectively. The fourth term indicates the subcapsule *control* (instance of *Control*). The last term is the definition of the *Handover* state diagram. Besides, the names *setCar*, *zgain1*, *zlose1*, *zgain2*, *zlose2* are used to represent the wired connections between the subcapsules in the π-calculus definitions. For example, the name *zgain1* allows the communication between the port *gain1* of *control* and the port *gain* of *trans1*, because the subcapsule's ports are renamed to the same name *zgain1*.

The Handover system is an example of mobile processes, where the messages received by a process in a communication can be used to rename its ports and, consequently, to allow the reconfiguration of its connections with other processes. Besides, the present example includes the need of the communication of more than one message at a time between processes, which emphasizes the importance of the polyadic π-calculus approach.

5. Comments and Conclusions

We propose a formal semantics to the UML-RT communicating elements through the mapping from the UML-RT to the polyadic π-calculus. The UML-RT is a UML profile widely used for the real-time system modeling, and the π-calculus is one of the main formalisms of the concurrent systems modeling. The proposed mapping from the UML-RT to the π-calculus is a practical approach of π-calculus, and it enables the use of formal methods during system development.

We described three examples. In the first example, we explained how to obtain the π-calculus definition of a UML-RT model using the proposed mapping rules. In the second example, we illustrated the possibility to formally verify system requirements using the π-calculus definitions of the model and some scenarios specified from the requirements. During the verification process, we showed that we can uncover some undesirable behavior, which are consequences of a poor system specification or an error introduced during the modeling phase. Finally, in the third example, we present the Handover protocol. This example demonstrates the importance of the reconfiguration of the connections between processes and the need of modeling of communication of more than one message at a time, which is well addressed using the polyadic π-calculus.

A real-time system designer who desires to formally reason about a UML-RT model can apply the proposed mapping rules on the UML-RT model to obtain the corresponding π-calculus definitions. Afterwards, the designer can manipulate the π-calculus definitions to explore possible execution paths and find situations that can cause functional problems or inconsistencies with functional requirements. Problems and inconsistencies include deadlock and nonsatisfaction of safety properties. This practice increases the confidence in the system development, because it anticipates the detection of problems in the modeling phase.

Two main contributions of the mapping proposed in this paper are due to polyadic π-calculus mobility, which is provided by name reconfiguration. The first contribution is the reuse of the capsule definition. For example, if a parent capsule has three instances of the same subcapsule, it is necessary to write one definition in the π-calculus for the subcapsule, and this definition is referenced three times by the parent capsule. It helps to reduce the number of the π-calculus definitions for the system. The second contribution of the name reconfiguration is the ability to represent the dynamic configuration of the unwired ports. It is important because a UML-RT capsule can rename an unwired port

during its state diagram execution, as it is shown in the first example.

Other contributions include the option for the polyadic π-calculus, which represents the communication of more than one message at a time. Besides, the proposed mapping deals with the following UML-RT concepts not yet addressed: the entry and the exit actions of a state, the composite states, and the transition chains.

In order to test the consistency of the mapping rules, we used the mapping to obtain π-calculus models of other UML models, besides the examples presented in this work. For instance, we have used the mapping on a system composed by a pair of semaphores to prove that two green lights of both semaphores cannot occur at the same time.

Future research directions include the improvement of the mapping rules to represent other UML-RT elements, such as history junction points, the guard conditions, and the capsule cardinalities. An interesting topic to investigate is to capture, from the UML-RT sequence diagram, the π-calculus definitions of the desirable scenarios to be met by the system under design. Another research issue is the study of formal approaches to represent the UML-RT timeouts, in order to be able to verify the time constraints of the modeled system.

References

[1] O. R. Ribeiro, J. M. Fernandes, and L. F. Pinto, "Model checking embedded systems with PROMELA," in *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECS '05)*, pp. 378–385, Greenbelt, Md, USA, April 2005.

[2] L. A. Cortés, P. Eles, and Z. Peng, "Verification of embedded systems using a petri net based representation," in *Proceedings of the 13th International Symposium on System Level Synthesis (ISSS '00)*, pp. 149–155, Madrid, Spain, 2000.

[3] J. Baeten and J. A. Bergstra, "Six issues concerning future directions in concurrency research," *ACM Computing Surveys*, vol. 28, 1996.

[4] J. P. Bowen and M. G. Hinchey, "Seven more myths of formal methods," *IEEE Software*, vol. 12, no. 4, pp. 34–41, 1995.

[5] M. Thomas, "Formal methods and their role in developing safe systems," *High Integrity Systems Journal*, vol. 1, no. 5, pp. 447–451, 1996.

[6] RTCA, Incorporated. Software Considerations in Airborne Systems and Equipment Certification (DO178-B).

[7] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, Pearson, Delhi, India, 2005.

[8] OMG Unified Modeling Language Specification version 1.5. OMG document formal/2003-03-01, 2003.

[9] A. Gherbi and F. Khendek, "UML profiles for real-time systems and their applications," *Journal of Object Technology*, vol. 5, no. 4, pp. 149–169, 2006.

[10] L. Bichler, A. Radermacher, and A. Schürr, "Evaluating UML extensions for modeling real-time systems," in *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '02)*, p. 271.1, IEEE Computer Society, San Diego, Calif, USA, January 2002.

[11] B. Selic and J. Rumbaugh, *Using UML for Modeling RealTime Systems*, Rational Software Corporation, 1998.

[12] IBM Rational Rose RealTime Tool. Version 6.5.

[13] OMG Unified Modeling Language Specification version 2.1.2. OMG document formal/2007-11-04, 2007.

[14] B. Selic, "UML 2: a model-driven development tool," *IBM Systems Journal*, vol. 45, no. 3, pp. 607–620, 2006.

[15] IBM Rational Rose Technical Developer Tool, Version 7.0. Product specifications, http://www.developers.net/ibmshowcase/view/1847.

[16] J. A. H. Terriza, K. B. Akhlaki, and M. I. C. Tuñón, "Combining the description features of UML-RT and CSP+T specifications applied to a complete design of real-time systems," *International Journal of Information Technology*, vol. 2, no. 3, 2005.

[17] C. Fischer, E.-R. Olderog, and H. Wehrheim, "A CSP view on UML-RT structure diagrams," in *Proceedings of Fundamental Approaches to Software Engineering (FASE '01)*, 2001.

[18] G. Engels, J. M. Küster, R. Heckel, et al., "A methodology for specifying and analyzing consistency of object-oriented behavioral models," in *Proceedings of the 8th European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, Austria, 2001.

[19] R. Ramos, A. Sampaio, and A. Mota, "A semantics for UML-RT active classes via mapping into circus," in *Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '05)*, vol. 3535 of *Lecture Notes in Computer Science*, pp. 99–114, Athens, June 2005.

[20] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1985.

[21] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1989.

[22] A. Cavalcanti, A. Sampaio, and J. Woodcock, "Refinement in circus," in *Proceedings of the International Symposium of Formal Methods Europe*, vol. 2391 of *Lecture Notes In Computer Science*, Springer, 2002.

[23] A. Knapp, S. Merz, and C. Rauh, "Model checking timed UML state machines and collaborations," in *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, vol. 2469 of *Lecture Notes In Computer Science*, pp. 395–416, 2002.

[24] R. Milner, *Communicating and Mobile Systems: The π-Calculus*, Cambridge University Press, Cambridge, UK, 1999.

[25] J. M. Bezerra and C. M. Hirata, "A semantics for UML-RT using π-calculus," in *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP '07)*, pp. 75–82, Porto Alegre, Brazil, May 2007.

[26] G.-L. Ferrari, S. Gnesi, U. Montanari, and M. Pistore, "A model-checking verification environment for mobile processes," *ACM Transactions on Software Engineering and Methodology*, vol. 12, no. 4, pp. 440–473, 2003.

[27] J. Parrow, "An introduction to the pi-calculus," in *Handbook of Process Algebra*, Bergstra, Ponse, and Smolka, Eds., pp. 479–543, Elsevier Science, Amsterdam, The Netherlands, 2001.

[28] L. Wischik, "New directions in implementing the pi-calculus," in *Cabernet Radicals Workshop*, 2002.

[29] S. D. Zilio, "Mobile processes: a commented bibliography," in *Proceedings of the 4th Summer School on Modelling and Verification of Parallel Processes*, vol. 2067 of *Lecture Notes in Computer Science*, pp. 206–222, Springer, 2001.

[30] M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1992.

[31] G. Smith, *The Object-Z Specification Language*, Advances in Formal Methods Series, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[32] H. Miao, L. Liu, and L. Li, "Formalizing UML models with Object-Z," in *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, pp. 523–534, Springer, 2002.

[33] S. Kim and D. Carrington, "A formal model of the UML metamodel: the UML state machine and its integrity constraints," in *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pp. 497–516, Springer, 2002.

[34] K. Taguchi, J. S. Dong, and G. Ciobanu, "Relating π-calculus to Object-Z," in *Proceedings of the 9th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '04)*, pp. 97–106, Florence, Italy, April 2004.

[35] K. Taguchi and J. Dong, "An overview of mobile Object-Z," in *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, pp. 144–155, Springer, 2002.

[36] A. Rarau and K. Pusztai, "An experience with using Z for mobile computing," in *Proceedings of Software Engineering and Applications*, Cambridge, Mass, USA, 2002.