A Semantics for UML-RT Active Classes via Mapping into Circus

Rodrigo Ramos, Augusto Sampaio and Alexandre Mota

Informatics Center, Federal University of Pernambuco
P.O.Box 7851 - 50.740-540, Recife-PE, Brazil
{rtr,acas,acm}@cin.ufpe.br

Abstract. The lack of a formal semantics for UML-RT makes it inadequate for rigourous system development, especially if the preservation of behaviour is a major concern when applying well-known model transformations, like refactorings and refinements. In this paper, we propose a semantics for UML-RT active classes through a mapping into *Circus*, a specification language that combines CSP, Z and specification statements. As a consequence of the translation, we are able to prove that model transformations preserve both static and dynamic behaviour, using refinement laws and a relational semantics of *Circus*, based on the Unifying Theories of Programming.

Keywords: UML-RT, *Circus*, method integration, model transformations

1 Introduction

As other object-oriented (OO) methods, UML [1] has tremendously influenced the software engineering modeling practice with rich structuring mechanisms. Despite its strengths, the rigorous development of non-trivial applications does not seem feasible without a formal semantics. The reason is that well-known model transformations might not preserve behaviour. This problem is even more serious in a model driven development, where transformations are as important as models, and involve different model views.

In the literature, several efforts address the problem through the integration of UML models with formal languages; there are some approaches to specification [2,3], like combinations of UML with Z [4] or with CSP [5]. A formalisation of class diagrams using the state-based Z notation is presented in [2]; a detailed comparison of general integration approaches involving Z and Object-oriented extensions of Z is discussed in [6]. Concerning formal semantics of state diagrams, a mapping into CSP processes is presented in [3]; typically, each contribution to formalise UML tends to concentrate on a single view (like state or class diagrams). Although some works like, for instance, [7], use a uniform notation to describe a mapping that considers both structure and behaviour, the presentation is informal, based on examples.

Similar limitations can be found in works [8,9] that formalise UML-RT [10] (a conservative UML profile that includes active objects to describe concurrent

and distributed applications) using CSP. They focus on the translation of the UML-RT structural view into CSP [8], and consider the behavioural representation of active objects [9] only partially. For example, statecharts of capules and protocols, and their relationship with the structure diagrams are not addressed. Also, model transformations are usually neglected or, when considered, the presentation is informal [11] and does not encompass all the model views.

In this paper, we propose a semantics for UML-RT via mapping into *Circus* [12], a language that combines CSP, Z and specification statements. We focus on mapping the new elements (active classes and other related constructs) that UML-RT adds to UML. We consider the following views of a UML-RT model in an integrated way: class, state and structure diagrams. We propose and prove a decomposition law for active classes, as an illustration of a sound model transformation; the law and its proof consider the views mentioned above.

One reason for using *Circus* is that its semantics is defined in the setting of the Unifying Theories of Programming [13]; this relational model has proved convenient for reasoning. Another advantage is that *Circus* includes the main design concepts of UML-RT. Unlike, for instance, CSP-OZ [14], *Circus* decouples event occurrences from state operations (like in UML-RT), and it has been designed to support a refinement calculus [15]; the decoupling seems crucial for addressing refinement. The laws of *Circus* have been inspiring both to propose laws for UML-RT ([16]) and to prove such laws as illustrated in this paper.

The next section gives an overview of UML-RT, and Section 3 introduces *Circus*. Section 4 presents a mapping from UML-RT into *Circus*. Soundness of model transformations is addressed in Section 5, where we propose and prove a transformation law. Finally, we summarise our results and topics for future work in Section 6.

2 UML-RT

The specification and design of a distributed system is a complex task involving data, behaviour, intercommunication and architectural aspects of the model. In order to fulfill these requirements, UML and ROOM (Real-Time Object-Oriented Modeling language) have been combined into UML for Real-Time (UML-RT) [10]. Some of the ROOM techniques also motivated extensions in the UML 2.0 [17] version. Here we use UML-RT because we consider that the proposed model for active objects is more consolidated than that proposed for UML 2.0. Furthermore, UML-RT counts with commercial tool support.

Using stereotype mechanisms, UML-RT introduces four new constructors: capsule, protocol, port and connector. Capsules (active classes) describe, potentially concurrent, architectural components that may interact with their environment only through signal-based objects called ports. Ports realise protocols, which define a set of signals that a capsule can receive or send. A protocol also defines the valid flow of information (signals) between connected ports of capsules. Connectors act as a physical communication channel between ports.

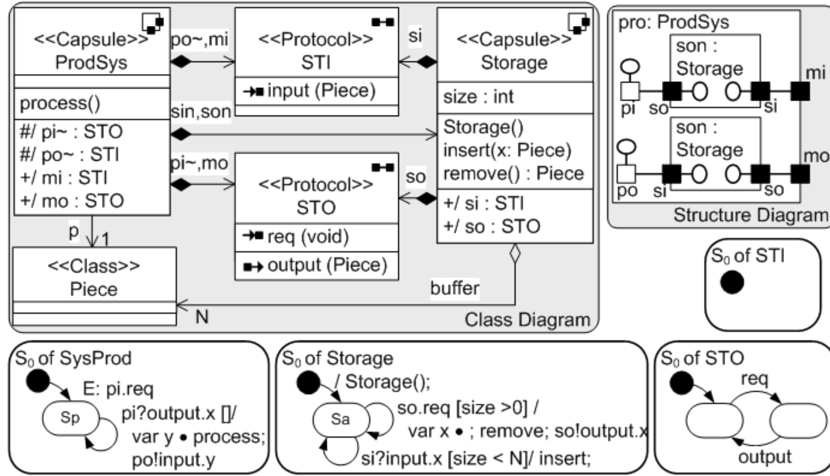


Fig. 1. Class, Structure and State Diagrams of a Manufacturing System

A UML-RT model of the system is formed of a set of diagrams and system properties. We choose diagrams that mainly represent the following architectural views: static data, dynamic behaviour, and dynamic object relationships; these are expressed, respectively, by class, state and structure diagrams. We directly express the system properties by invariants, pre- and post-conditions in *Circus*; they could alternatively be expressed in OCL [18], but an OCL to *Circus* mapping is out of the scope of this work.

In order to illustrate the notation, an example of a simplified manufacturing system is used. In Figure 1 (top left rectangle) a class diagram is presented. Capsules and protocols are graphically represented in the diagram by a box with a stereotype Capsule and Protocol, respectively; a symbol at the top right is also used to characterise their representations. The diagram emphasises the relationships between the ProdSys and Storage capsules. The capsule Storage is a bounded, reactive, *FIFO* buffer that is used to store objects of class Piece, and ProdSys is used to process these objects. These capsules have an association to the protocols, which are used to govern the communication among capsules. The protocol STO declares the input signal req (used to communicate the request of a work piece) and the output signal output (which communicates the delivery of a work piece). The protocol STI declares a signal input to store a piece.

By their own nature, capsules provide a high degree of information hiding. As the communication mechanism is via message passing, all capsule elements are hidden, including not only attributes, but also methods. The capsule Storage has a set of Pieces, represented by the association buffer, and methods to threat this set. The only visible elements in the capsule are ports, which can be connected to other capsule ports to establish communication; here, we assume that this communication is synchronous. This decoupling makes capsules highly reusable. In addition, a capsule can also be defined hierarchically, in terms of subcapsules, each with a state-machine and possibly a hierarchy of compound capsules.

A structure diagram, a kind of collaboration diagram, describes a capsule structural decomposition, showing the capsule interaction through connections among its ports and permitting hierarchically composed models. The structural decomposition of ProdSys is shown in Figure 1 (top right rectangle). It is composed by the capsule instances sin and son of Storage, which also communicate with ports of ProdSys; these capsule instances are created as a consequence of the association of ProdSys with the capsule Storage in the class diagram. Block filled squares in the capsule instances represent their ports, where white ones are conjugated (their signal directions are inverted in relation to the ordinary ports). Ports are basically divided into two types: relay ports and end ports. End ports are used by the capsule statechart to receive or transmit signals, while relay ports are used to connect other ports (like protected ports of the capsule or public ports of subcapsules) in the capsule internal structure diagram to the external environment. In the structure diagram of ProdSys, the ports pi and po are protected end ports used only by its statechart, while mi and mo are public relay ports used only to connect ports of subcapsules to the environment. In Storage, si and so are public end ports.

The capsule behaviour is described in terms of UML-RT statecharts, which extend the ROOMCharts concept [19] and differ from the standard UML statecharts [1] including some adaptations to better describe active objects (for instance, UML-RT statechart do not have final states). A statechart is composed by transitions and states; in general, a transition has the form `p.e[g]/a`, where e is an input signal, p is the port through which the signal arrives, g is a guard and a is an action. Input signals and a true guard trigger the transition. As a result, the corresponding action is executed.

We assume that events, guards and actions are expressed using the *Circus* notation. For example, in the statechart of Storage, there are two transitions from state Sa. The one on the right triggers if the req signal arrives through port so and the buffer is non-empty. The corresponding action declares a variable x to capture return of the method remove. This is the way it is done in *Circus*, since remove is actually interpreted as a Z-Schema, as explained in the next section. The value of x is then through channel so. The syntax for writing these actions related to communication are also as in *Circus*. In this work we do not consider capsule inheritance, mainly because its semantics in UML-RT is not yet well-defined; according to the ROOMCharts definitions [19], inherited capsule is able to arbitrarily change the behaviour defined in its super capsule.

In UML-RT, states are classified as initial, choice, composite or simple. Initial states are transient states that mark the starting point of a state-machine. A composite state groups other states, whereas a simple one has no other state inside. Choice states are those that involve a decision of which path to take according to its guard; there exist only two outgoing transitions: one is triggered when the guard is true; otherwise the second is triggered. Composite states are divided into two types: Or-States and And-States. An Or-State defines that only one of its substates is active, whereas And-States contain regions that are

executed in parallel, and permit each region to have an active state per time. Further details about composite states are presented in Section 4.2.

For a capsule state-machine, we assume that there is a composite state that contains all other states; it is called the top state (S_0) and is implicitly reached when the capsule instance is created. Figure 1 (bottom) presents the statecharts of ProdSys, Storage, STO and STI. Their respective top states (S_0) are Or-States that sequentially execute the inner states. In Storage, for example, the initial state moves to the state Sa. Next it waits the arrival of the input or req signals. The behaviour of Storage is further explained in the next section.

3 *Circus*

The language ***Circus*** [12] includes the notion of a process, whose state is defined using a Z schema and behaviour by an action expressed in the CSP notation. Process interaction is defined via channels to communicate values or just synchronise events. Therefore, the specification of a ***Circus*** program is defined by a sequence of paragraphs, which can include: Z paragraphs, process definitions, and channel declarations. As a simple example, consider the capsule Storate of Figure 1. It can be specified in ***Circus*** as the following process.

$| \; N : \mathbb{N}$
$T_{\mathsf{STI}} ::= \mathsf{input} \ll \mathsf{Piece} \gg$
$\textbf{channel}\ \mathsf{si} : T_{\mathsf{STI}}$
$T_{\mathsf{STO}} ::= \mathsf{req} \mid \mathsf{output} \ll \mathsf{Piece} \gg$
$\textbf{channel}\ \mathsf{so} : T_{\mathsf{STO}}$
$\textbf{process}\ \mathsf{Storage} \mathrel{\widehat=} \textbf{begin}$
$\quad \textbf{state}\ \mathit{StorageState} \mathrel{\widehat=} [\mathsf{buff} : \mathrm{seq}\,\mathsf{Piece};\ \mathsf{size} : 0..N \mid \mathsf{size} = \#\mathsf{buff} \leq N]$
$\quad \textbf{initial}\ \mathit{StorageInit} \mathrel{\widehat=} [\mathit{StorageState}' \mid \mathsf{buff}' = \langle\rangle \land \mathsf{size}' = 0]$
$\quad \mathsf{insert} \mathrel{\widehat=} [\Delta \mathit{StorageState};\ x? : \mathsf{Piece} \mid \mathsf{size} < N \land$
$\quad\quad \mathsf{buff}' = \mathsf{buff} \frown \langle x? \rangle \land \mathsf{size}' = \mathsf{size} + 1]$
$\quad \mathsf{remove} \mathrel{\widehat=} [\Delta \mathit{StorageState};\ x! : \mathsf{Piece} \mid \mathsf{size} > 0 \land x! = \mathit{head}\,\mathsf{buff} \land$
$\quad\quad \mathsf{buff}' = \mathit{tail}\,\mathsf{buff} \land \mathsf{size}' = \mathsf{size} - 1]$
$\quad \mathsf{Sa} \mathrel{\widehat=} (\mathsf{size} < N \,\&\, \mathsf{si?input.x} \rightarrow \mathsf{insert};\ \mathsf{Sa})$
$\quad\quad \Box\ (\mathsf{size} > 0 \,\&\, \mathsf{so.req} \rightarrow (\textbf{var}\,\mathsf{x} : \mathsf{Piece} \bullet \mathsf{remove};\ \mathsf{so!output.x});\ \mathsf{Sa})$
$\bullet\ \mathit{StorageInit};\ \mathit{Sa}$
$\textbf{end}$

The maximum size of this buffer is a strictly positive constant N. The buffer program takes its inputs and supplies its outputs through the channels si and so, respectively. The free types T_{STI} and T_{STO} categorise the values communicated by these channels. In T_{STO}, req and output are constructors of the free type, used to communicate the request and delivery of objects, respectively. Similarly, in T_{STI}, input is used to represent the storage of objects of type Piece.

In ***Circus*** the body of a declaration of a process is delimited by the **begin** and **end** keywords; it is composed by a sequence of paragraphs and a main action (after the • symbol), which defines the process behaviour. One of these paragraphs is used to describe the state of the process (identified by the keyword **state**), which encapsulates its data components. Furthermore, the other

paragraphs describe process operations and actions, that are used to structure the process specification and the main action.

In our example, the process Storage encapsulates two state components in the Z schema *BufferState*: an ordered list buffer of contents and the size of this list, represented by size. Initially, the buffer is empty and, therefore, its size is zero; this is specified as a state initialisation action *StorageInit*.

The main action initialises the buffer and repeatedly offers the choice of input and req. The signal input is guarded by size $< N$. The process accepts an input whenever there is space to store the new value; in this case, the piece is appended to the bounded sequence and the size incremented. The effect on the state is described by a schema as usual, using the operator insert. The action si?input.x $\rightarrow$ put is a prefixing in the style of CSP. A new input variable x is introduced, and a value input through the channel si is assigned to it. Afterwards, the action insert is executed. The req signal is enabled providing that the buffer contains something (size > 0). Afterwards, the Buffer offers output to deliver a piece. The associated state modification is defined by the remove schema action, which removes the head of buffer and updates the size accordingly.

The example has shown how processes are constructed from actions, but processes may themselves be combined with CSP operators, such as parallel composition ($A_1 \llbracket cs \rrbracket A_2$). The meaning of a new process constructed in this way is obtained from the conjunction of the constituent states of the processes in composition (A_1 and A_2) and the parallel combination of their main actions, synchronising on the set of channels (cs).

At the level of actions, the Circus parallel operator is actually slightly different from that of CSP. To resolve conflicts in the access to the variables in scope, it requires a synchronisation set and two sets that partition those variables. In the parallelism $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, the actions A_1 and A_2 synchronise on the channels in set cs. Although both A_1 and A_2 have access to all variables, they can only modify the values of the variables in ns_1 and ns_2, respectively.

Further explanation of the ***Circus*** notation used in this work is introduced as the need arises, in the next section. A more detailed presentation of ***Circus***, including its complete grammar and formal semantics, can be found in [20].

4 Semantic Mapping

This section gives meaning to UML-RT elements through a mapping into ***Circus***. This translation provides a mapping of structural and behaviour UML-RT elements into ***Circus***.

In our approach, UML-RT classifiers with an associated behaviour (capsules and protocols) are mapped into processes, and ports into channels. The mapping of classes has been addressed by several authors, and is out of the scope of our work. For example, in [2] classes are mapped directly into Z paragraphs. In this sense, our work complements those approaches.

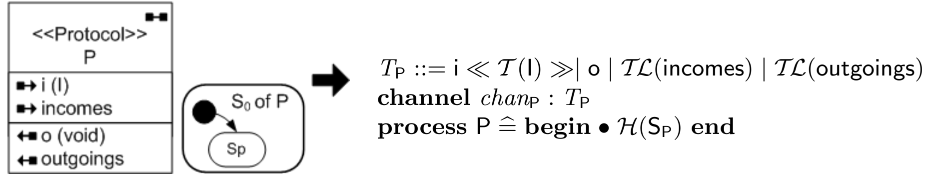
In our strategy the target of the translation is a ***Circus*** specification that is itself the meaning of the original model. Concerning the structure diagrams,

we consider that they implicitly define an extensional view of the system; they contain the set of capsule instances. To deal with hierarchical structures, we assume that all capsule instances, ports and connections have distinct names.

When mapping elements declared as a list (such as attributes and methods of a capsule, or signals of a protocol), by convention we single out one of its elements, present its mapping , and invoke a meta function ($\mathcal{TL}()$) to translate the remaining elements. We assume that there are overloaded definitions of $\mathcal{TL}()$ for each kind of list. In practice such lists can obviously be empty, but we avoid this trivial case assuming that they have at least one element.

4.1 UML-RT Structural Elements

A protocol declaration in UML-RT encapsulates both the communication elements and the behaviour. In *Circus*, this gives rise to two major elements: a stateless process that captures this behaviour and a channel to represent the communication elements. Regarding the signals, a possible mapping would be to introduce a channel associated with each signal. Rather, we use a single channel to communicate all signals of a protocol. This channel communicates values of a free type, with each constructor representing a signal. Using a single channel facilitates the mapping of capsules presented next. For instance, it is more convenient to use a single channel in contexts involving synchronisation or renaming.

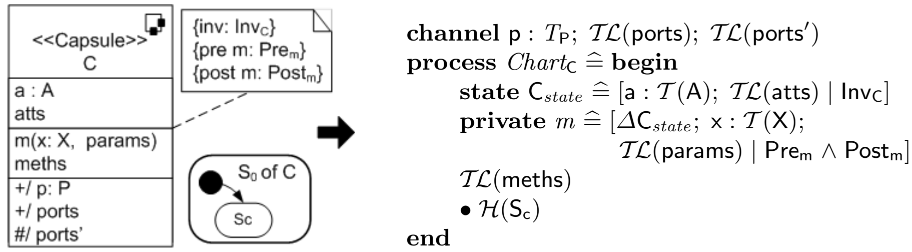


$$T_{\mathsf{P}} ::= \mathsf{i} \ll \mathcal{T}(\mathsf{I}) \gg | \ \mathsf{o} \ | \ \mathcal{TL}(\mathsf{incomes}) \ | \ \mathcal{TL}(\mathsf{outgoings})$$
$$\textbf{channel } chan_{\mathsf{P}} : T_{\mathsf{P}}$$
$$\textbf{process } \mathsf{P} \mathrel{\widehat=} \textbf{begin} \bullet \mathcal{H}(\mathsf{S_P}) \textbf{ end}$$

In names like $chan_{\mathsf{P}}$ above, we assume that P is a placeholder for the actual protocol name. In this way, the mapping of protocol STO in the example (Figure 1) generates a corresponding channel $chan_{\mathsf{STO}}$. The channel $chan_{\mathsf{P}}$ communicates values of the free type T_{P}; each value represents a signal. Parameterless signals, like the outgoing signal o above, are translated into constants; parameterised signals are mapped into data type constructors (like i). The type of the parameter is translated into a corresponding *Circus* type by function $\mathcal{T}()$. The remaining signals ($\mathsf{incomes}$ and $\mathsf{outgoings}$) are mapped by the function $\mathcal{TL}()$.

The behaviour of the protocol P is represented by $\mathcal{H}(\mathsf{S_P})$, where $\mathsf{S_P}$ stands for a state that encloses all other states of its statechart. The function $\mathcal{H}()$, which translates a statechart into a *Circus* action, is explained in Section 4.2.

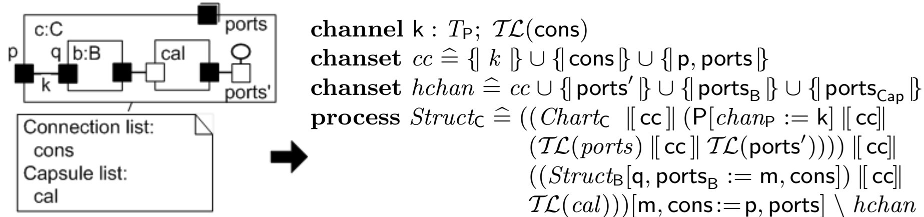
Capsules are also defined as processes, with methods defined as schema operations, and attributes mapped into a Z state schema (the process state). Each port generates a channel with the same type of the corresponding channel of the protocol, and has its behaviour described by the process obtained from the mapping of its protocol synchronised with that obtained from the capsule statechart. Observe that in UML-RT the type of a port is the protocol itself. In *Circus* the type of the channel originated from the port is the free type that represents the signals (as explained in the mapping of a protocol).



channel $\mathsf{p} : T_{\mathsf{P}};\ \mathcal{TL}(\mathsf{ports});\ \mathcal{TL}(\mathsf{ports}')$
process $Chart_{\mathsf{C}} \mathrel{\widehat=}$ **begin**
  state $\mathsf{C}_{state} \mathrel{\widehat=} [\mathsf{a} : \mathcal{T}(\mathsf{A});\ \mathcal{TL}(\mathsf{atts}) \mid \mathsf{Inv_C}]$
  private $m \mathrel{\widehat=} [\Delta \mathsf{C}_{state};\ \mathsf{x} : \mathcal{T}(\mathsf{X});$
    $\mathcal{TL}(\mathsf{params}) \mid \mathsf{Pre_m} \wedge \mathsf{Post_m}]$
  $\mathcal{TL}(\mathsf{meths})$
  $\bullet\ \mathcal{H}(\mathsf{S_c})$
end

In the above mapping, the process $Chart_{\mathsf{C}}$ deals with the views represented by class and state diagrams. It encapsulates all actions that manipulate the private attributes of the capsule C. In the capsule C above, the compartments correspond to attributes, methods and ports. Therefore, a, m and p are those that we single out. The attribute a is mapped to an attribute in the state of $Chart_{\mathsf{C}}$ with its corresponding type in *Circus*, given by $\mathcal{T}(\mathsf{A})$; the other attributes atts are mapped by the function $\mathcal{TL}()$, as previously explained. The invariant $\mathsf{Inv_C}$ cames from the UML-RT note element on the left, and it is assumed to be already described in *Circus*. The method m() is mapped to an operator that could change any state attribute and whose parameters are mapped into schema attributes, just like a has been included in the state schema; similarly the function $\mathcal{TL}()$ maps the other methods meths. Like the invariant $\mathsf{Inv_C}$, the pre- and post-conditions $\mathsf{Pre_m}$ land $\mathsf{Post_m}$ are written in *Circus*. The port p is mapped to a channel with the same type T_{P} of the channel $Chan_{\mathsf{P}}$ used by the protocol P. The main action of $Chart_{\mathsf{C}}$ is expressed by $\mathcal{H}(\mathsf{S_C})$, which represents the mapping of the statechart of capsule C, explained in Section 4.2.

We need also consider structure diagrams, dealt with by the mapping below. The process that deals with this view represents the observational behaviour of capsule C after considering the restrictions imposed by its ports to the corresponding communication channels used by the process, and considering the parallelism of all its connected subcapsules.



channel $\mathsf{k} : T_{\mathsf{P}};\ \mathcal{TL}(\mathsf{cons})$
chanset $cc \mathrel{\widehat=} \{\!|\ k\ |\!\} \cup \{\!|\ \mathsf{cons}\ |\!\} \cup \{\!|\ \mathsf{p}, \mathsf{ports}\ |\!\}$
chanset $hchan \mathrel{\widehat=} cc \cup \{\!|\ \mathsf{ports}'\ |\!\} \cup \{\!|\ \mathsf{ports_B}\ |\!\} \cup \{\!|\ \mathsf{ports_{Cap}}\ |\!\}$
process $Struct_{\mathsf{C}} \mathrel{\widehat=} ((Chart_{\mathsf{C}}\ [\![\ \mathsf{cc}\ ]\!]\ (\mathsf{P}[chan_{\mathsf{P}} := \mathsf{k}]\ [\![\ \mathsf{cc}\ ]\!]$
  $(\mathcal{TL}(ports)\ [\![\ \mathsf{cc}\ ]\!]\ \mathcal{TL}(\mathsf{ports}')))) \ [\![\ \mathsf{cc}\ ]\!]$
  $((Struct_{\mathsf{B}}[\mathsf{q}, \mathsf{ports_B} := \mathsf{m}, \mathsf{cons}])\ [\![\ \mathsf{cc}\ ]\!]$
  $\mathcal{TL}(cal)))[\mathsf{m}, \mathsf{cons} := \mathsf{p}, \mathsf{ports}] \setminus hchan$

In the process $(\mathsf{Struct_C})$, the behaviour of $Chart_{\mathsf{C}}$ is synchronised with the behaviour of all ports in capsule C; we single out here the port p, whose behaviour is represented by process P. The channel $chan_{\mathsf{P}}$ used by P has to be renamed with the name of the connector (k) that links the port p to another port in the system. The other ports are similarly mapped by the function $\mathcal{TL}()$. We single out the subcapsule B from the subcapsule list cal, which has all public ports q and $\mathsf{ports_B}$ renamed to their associated connections k and cons; $\mathsf{ports_B}$ stands for the public port list of B. After parallelising all capsules in the structure diagram of C, an injective function is needed to rename $([\mathsf{m}, \mathsf{cons} := \mathsf{p}, \mathsf{ports}])$ connections

to associated public ports, and hide ($\setminus$ $hchan$) connections protected port (and subcapsule port) channels.

Because we represent port connections through renaming and synchronisation, we need to assume that ports are connected in the lower structure level in which they are present; actually this is a good design practice.

4.2 UML-RT Behavioural Elements

Our mapping of capsule and protocol statecharts into *Circus* is based on the work reported in [3], which presents the formalisation of UML statecharts in CSP. Nevertheless, we extend [3] in order to consider *Circus* actions, deal with parallelism (And-States) and composite states with multiple initial points.

Let M be a state-machine, and S_M be the set of states of M. The set of events of M is denoted by E_M, and its actions and boolean guards by A_M and G_M, respectively. Furthermore, let SI_M be the set of all initial states, SCh_M the set of all choice states, SS_M the set of all simple states and SCo_M the set of all composite states of S_M.

As a statechart can be identified by the topmost state that contains inner states, mapping a statechart reduces to mapping a state. Thus, let $\mathcal{H}$ be a function that takes a state and yields its *Circus* representation.

$\mathcal{H} : S_M \rightarrow CSPAction$

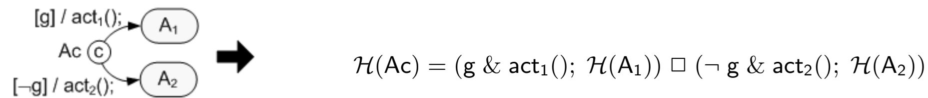
We assume that an action in A_M is expressed as a method call and, therefore, does not need be translated. Like other predicates, guards in A_M are written using the *Circus* syntax. As ports have an associated channel in *Circus*, with the same name, and signals are expressed as values of this channel type, a signal e of a port p can be directly written as an event pattern matching in *Circus*.

Every pattern to which the mapping function $\mathcal{H}()$ is applied gives rise to a separate mapping rule. On the left of each rule, we illustrate the pattern as a template statechart.

The first pattern address initial states, which have only one outgoing transition, and no entry nor exit actions (actions executed before the state becomes active and inactive, respectively). Let Ai be an initial state, with Ai $\in SI_M$, act() the action of its outgoing transition, and $\mathsf{A_1}$ the target of this transition, then:

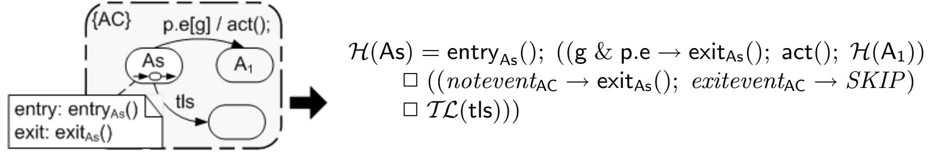
$\mathcal{H}(\mathsf{Ai}) = \mathsf{act}();\ \mathcal{H}(\mathsf{A_1})$

For a choice state, its translation is as follows. Let Ac by a choice state in SCh_M, which has only two outgoing transitions and one guard g (these transitions have no events and are triggered depending on the evaluation of g), then:



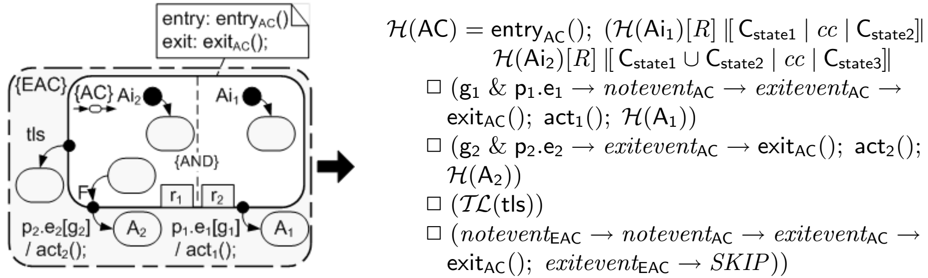
$\mathcal{H}(\mathsf{Ac}) = (\mathsf{g}\ \&\ \mathsf{act_1}();\ \mathcal{H}(\mathsf{A_1}))\ \Box\ (\neg\ \mathsf{g}\ \&\ \mathsf{act_2}();\ \mathcal{H}(\mathsf{A_2}))$

Now suppose the translation of a simple state. Let As be a simple state in SS_M, AC the composite state in SCo_M that encloses it and tls the outgoing transitions of As (where p.e[g]/act is singled out), then:



When the state As is active, it executes the entry action $\mathsf{entry_{As}}()$. Then, it waits for a new signal to be communicated by the environment (the external choice captures this decision). Such an event p.e can only be performed if its guard g is satisfied. Finally the exit action $\mathsf{exit_{As}}()$ and the action act() associated to the trigged transition are executed. The remaining transitions in tls are mapped by the function $\mathcal{TL}()$, as previously explained. Further the simple state As can exit if an outgoing (group) transition of its enclosing state AC is triggered; in this case, a notify event $\mathit{notevent}_{\mathsf{AC}}$ is sent by AC to ensure that the exit action of As is executed before its exit action. After the execution of the As exit action, an event $\mathit{exitevent}_{\mathsf{AC}}$ is launched to allow the execution of the exit action of AC.

To formalise composite states, we present the translation of a composite And-State with its concurrent regions formed by Or-States. In this scenario, the translation of a composite Or-State is given by an And-State with a unique region. Let AC and EAC be composite states in SCo_M (where EAC encloses AC), Ai_1 and Ai_2 the initial states in SI_M of the regions of AC, tls the outgoing transitions of AC (where $\mathsf{p}_1.\mathsf{e}_1[\mathsf{g}_1]/\mathsf{act}_1$ and $\mathsf{p}_2.\mathsf{e}_2[\mathsf{g}_2]/\mathsf{act}_2$ are singled out) and $[R]$ a renaming function that replaces channels associated to connected ports for the respective channels associated to their connections (see Section 4.1), then:



When the And-State AC is active, it executes the entry action $\mathit{entry}_{\mathsf{AC}}()$. Then, it executes the initial states of each region. These regions can synchronise their events through the channel set cc, which are the internal connections of the capsule that own the statechart. Each region r_1 and r_2 can only modify the values of the variables in $\mathsf{C_{state1}}$ and $\mathsf{C_{state2}}$, respectively; the exit action of AC and the action of its transitions can only modify the values in $\mathsf{C_{state3}}$. The capsule state contains the union of $\mathsf{C_{state1}}$, $\mathsf{C_{state2}}$ and $\mathsf{C_{state3}}$. The enclosed states in each region of AC are sequentially reached. At any moment, the state AC can receive an event $\mathsf{p}_1.\mathsf{e}_1$ that triggers a group transition (a transition that emanates directly from the border of the composite state and interrupts the state and substates in any situation, however allowing that they normally finish the execution of their exit-actions). When a group transition is triggered, the event $\mathit{notevent}_{\mathsf{AC}}$ occurs, and

AC waits for the active inner state to execute its exit action. When a notification $exitevent_{\mathsf{AC}}$ arrives, the state AC is able to execute its exit action $\mathsf{exit}_{\mathsf{AC}}$ and the transition action act_1; then it moves to the next state. Transitions that emanate from an enclosed state through a junction point, like the one triggered by $\mathsf{p}_2.\mathsf{e}_2$, behave similarly, but do not send a $notevent_{\mathsf{AC}}$ event. This type of junction is mapped to a transient inner state of AC; for a junction point F, its translation corresponds to $\mathcal{H}(F) = exitevent_{\mathsf{AC}} \rightarrow SKIP$. The other outgoing transitions tls of AC can be similarly translated by the function $\mathcal{TL}()$.

As simple states, a composite state AC can be enclosed by another composite state EAC. In this situation, AC receives the notification $notevent_{\mathsf{EAC}}$ to indicate the arrival of an event in EAC. When this happens, AC must send $notevent_{\mathsf{AC}}$ to its immediate active inner state to ensure that all active enclosed states execute their exit action before the AC exit action. After AC executes $\mathsf{exit}_{\mathsf{AC}}()$, $exitevent_{\mathsf{EAC}}$ is sent to allow EAC to execute its exit action.

For simplicity, we do not consider here history states.

5 Model Transformations and Soundness

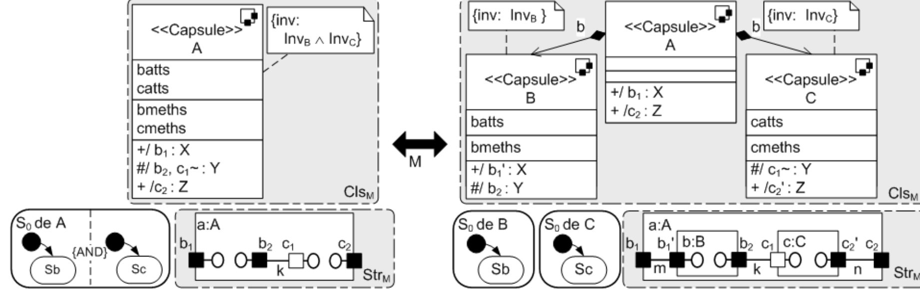
Translating UML-RT models into *Circus* can then benefit from the formal semantics as well as the refinement notions and laws of *Circus*. For example, the meaning of capsule refinement is interpreted as process refinement in *Circus*. Similarly, action refinement can be used to capture statechart transformations.

As illustration, we present one transformation law that involves the three most important diagrams of UML-RT: state, class and structure diagrams. As previously mentioned, we consider structure diagrams because they represent some relations between capsules that are not expressed in class diagrams; so they must be taken into account to preserve the model behaviour.

Law 1 decomposes a capsule A into parallel component capsules (B and C) in order to tackle design complexity and to potentially improve reuse. The side condition requires that A must be partitioned, a concept that is explained next.

On the left-hand side of Law 1 the state machine of A is an And-State composed of two states (Sb and Sc), which may interact (internal communication) through the conjugated ports b_2 and c_1 (as captured by the structure diagram on the left-hand side). The other two ports (b_1 and c_2) are used for external communication by states Sb and Sc, respectively. Furthermore, in transitions on Sb, only the attributes batts and the methods bmeths (that may reference only the attributes batts) are used; analogously, transitions of Sc use only the attributes catts and the methods cmeths (that may reference only the attributes catts). Finally, the invariant of A is the conjunction $\mathsf{Inv}_{\mathsf{B}} \wedge \mathsf{Inv}_{\mathsf{C}}$, where $\mathsf{Inv}_{\mathsf{B}}$ involves only batts as free variables, and $\mathsf{Inv}_{\mathsf{C}}$ only catts. When a capsule obeys such conditions, we say that it is partitioned. In this case, there are two partitions: one is $\langle \mathsf{batts}, \mathsf{Inv}_{\mathsf{B}}, \mathsf{bmeths}, (\mathsf{b}_1, \mathsf{b}_2), \mathsf{Sb} \rangle$ and the other is $\langle \mathsf{catts}, \mathsf{Inv}_{\mathsf{C}}, \mathsf{cmeths}, (\mathsf{c}_1, \mathsf{c}_2), \mathsf{Sc} \rangle$.

Law 1. *Capsule Decomposition*



provided $\langle \mathsf{batts}, \mathsf{Inv_B}, \mathsf{bmeths}, (\mathsf{b}_1, \mathsf{b}_2), \mathsf{Sb} \rangle$ *and* $\langle \mathsf{catts}, \mathsf{Inv_C}, \mathsf{cmeths}, (\mathsf{c}_1, \mathsf{c}_2), \mathsf{Sc} \rangle$ *partition A; The statecharts of the protocol* X *and* Z *are deterministic.*

The effect of the decomposition is to create two new component capsules, B and C, one for each partition, and redesign the original capsule A to act as a mediator. In general, the new behaviour of A might depend on the particular form of decomposition. Law 1 captures a parallel decomposition. On the right-hand side of the law, A has no state machine. It completely delegates its original behaviour to B and C through the structure diagram.

Concerning the structure diagram on the right-hand side of the law, it shows how A encapsulates B and C. When A is created, it automatically creates the instances of B and C, which execute concurrently. The public ports b_1 and c_2 are preserved in A. Capsule B has as its public port an image of b_1, called b'_1. Although this port is public in B, it is only visible inside the structure diagram of A. The role of this port is to allow B to receive the external signals received from A through port b_1, as captured by the connection between b'_1 and b_1 in the structure diagram of A. Analogously, c_2 and b'_2 have the same relationship, concerning capsules A and C. The internal ports b_2 and c_1 are moved to capsules B and C, respectively, and play the same role as before.

5.1 Soundness

Based on the semantic mapping presented in Section 4, we can translate the two sides of Law 1 and check its validity. Instead of appealing directly to the semantics of *Circus* we can use its refinement laws [12] to carry out the proof. Actually, Law 1 has been inspired by the following law presented for *Circus*. Obviously, when there are no corresponding laws in *Circus*, it might be necessary to carry out the proof directly in the *Circus* UTP semantics. Even when there is a corresponding law in *Circus*, we will see that the proof is more elaborate than using just this law, since we consider structure diagrams.

Law 2. *Process splitting*

Let qd and rd stand for the declarations of the processes Q and R, determined by Q.st, Q.pps, and Q.act, and R.st, R.pps, and R.act, respectively; let F stand for a context which must also make sense as a function on processes. Then

$$\begin{array}{lcl}\textbf{process}\ P \mathrel{\widehat=} \textbf{begin} & = & (qd\ rd\ \textbf{process} P \mathrel{\widehat=} F(Q,R)) \\ \quad \textbf{state}\ State \mathrel{\widehat=} Q.st \wedge R.st & & \\ \quad Q.pps \uparrow R.st & & \\ \quad R.pps \uparrow Q.st & & \\ \quad \bullet\ F(Q.act, R.act)\ \textbf{end} & & \end{array}$$

provided *Q.pps and R.pps are disjoint with respect to R.st and Q.st.* □

In Law 2, the state of P is defined as the conjunction of two other state schemas: $Q.st$ and $R.st$. The actions of P are $Q.pps \uparrow R.st$ and $R.pps \uparrow Q.st$, which handle the partitions of the state separately. In $Q.pps \uparrow R.st$, each schema expression of $Q.pps$ is conjoined with $\Xi R.st$; this means that the state components of $R.st$ do not change (similarly for $R.pps \uparrow Q.st$). Two sets of process paragraphs pps_1 and pps_2 are disjoint with respect to states s_1 and s_2 if, and only if, no command nor CSP action expression in pps_1 refers to components of s_2 or to paragraph names in pps_2; similarly, for pps_2 and components of s_2.

Proof of Law 1. First we deal with the class and the state diagrams. The mapping of these views of capsule A as a *Circus* process is obtained by using the second mapping rule presented in Section 4.1 and the last rule of Section 4.2. We write two partitions using the operator ↑, as in Law 2. Strictly, the attributes and methods *batts*, *catts*, *bmeths* and *cmeths* must be mapped using the function $\mathcal{TL}()$; here we omit its application for a matter of readability.

$$\begin{array}{l}\textbf{process}\ Chart_{\mathsf{A}} \mathrel{\widehat=} \textbf{begin} \\ \quad \textbf{state}\ State \mathrel{\widehat=} [\mathsf{batts} \wedge \mathsf{catts} \mid \mathsf{Inv_B} \wedge \mathsf{Inv_C}] \\ \quad \mathsf{bmeths} \uparrow \mathsf{catts} \\ \quad \mathsf{cmeths} \uparrow \mathsf{batts} \\ \quad \bullet\ \mathcal{H}(\mathsf{S_B})[\mathsf{b_2} := \mathsf{k}] \ [\![\mathsf{batts} \mid \{\!|\ \mathsf{k}, \mathsf{b_1}, \mathsf{c_2}\ |\!\} \mid \mathsf{catts}]\!] \mathcal{H}(\mathsf{S_C})[\mathsf{c_1} := \mathsf{k}] \\ \textbf{end}\end{array}$$

From Law 2, the following equality holds:

$$Chart_{\mathsf{A}} = Chart_{\mathsf{B}}[\mathsf{b_2} := \mathsf{k}] \ [\![\{\!|\ \mathsf{k}, \mathsf{b_1}, \mathsf{c_2}\ |\!\}]\!] \ Chart_{\mathsf{C}}[\mathsf{c_1} := \mathsf{k}]$$

where $Chart_{\mathsf{B}}$ and $Chart_{\mathsf{B}}$ are declared as:

$$\textbf{process}\ Chart_{\mathsf{B}} \mathrel{\widehat=} \textbf{begin state}\ State \mathrel{\widehat=} [\mathsf{batts} \mid \mathsf{Inv_B}]\ \mathsf{bmeths} \bullet \mathcal{H}(\mathsf{S_B})\ \textbf{end}$$
$$\textbf{process}\ Chart_{\mathsf{C}} \mathrel{\widehat=} \textbf{begin state}\ State \mathrel{\widehat=} [\mathsf{catts} \mid \mathsf{Inv_C}]\ \mathsf{cmeths} \bullet \mathcal{H}(\mathsf{S_C})\ \textbf{end}$$

Now we consider the structural part of capsule A. This is mapped into the *Circus* process $Struct_{\mathsf{A}}$, presented below.

$$\begin{array}{l} Struct_{\mathsf{A}} \mathrel{\widehat=} (Chart_{\mathsf{A}}[\mathsf{b_2}, \mathsf{c_1} := \mathsf{k}, \mathsf{k}] \ [\![\{\!| \mathsf{k}, \mathsf{b_1}, \mathsf{c_2} |\!\}]\!] \ \mathsf{X}[chan_{\mathsf{X}} := \mathsf{b_1}] \ [\![\{\!| \mathsf{k}, \mathsf{b_1}, \mathsf{c_2} |\!\}]\!] \\ \quad \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{k}] \ [\![\{\!| \mathsf{k}, \mathsf{b_1}, \mathsf{c_2} |\!\}]\!] \ \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{k}] \ [\![\{\!| \mathsf{k}, \mathsf{b_1}, \mathsf{c_2} |\!\}]\!] \ \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c_2}]) \setminus \{\!| \mathsf{k} |\!\} \end{array}$$

The structure diagrams of B and C, as well as that of A on the right-hand side of Law 1, are mapped similarly. To avoid confusion between the two occurrences of A in the law, we refer to the occurrence on the left simply as A, and that on the right as A′. Therefore, we want to prove that $Struct_{\mathsf{A}}$ has the same behaviour as $Struct_{\mathsf{A'}}$, where the latter is composed of the processes $Struct_{\mathsf{B}}$ and $Struct_{\mathsf{C}}$:

$$\begin{array}{l} Struct'_{\mathsf{A}} \mathrel{\widehat=} ((Struct_{\mathsf{B}}[\mathsf{b'}_1, \mathsf{b_2} := \mathsf{m}, k] \ [\![\{\!|\ \mathsf{k}\ |\!\}]\!] \ Struct_{\mathsf{C}}[\mathsf{c'}_2, \mathsf{c_1} := \mathsf{n}, k]) \ [\![\{\!|\ k\ |\!\}]\!] \\ \quad (\mathsf{X}[chan_{\mathsf{X}} := \mathsf{m}] \ [\![\{\!|\ k\ |\!\}]\!] \ \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{n}]))[\mathsf{m}, \mathsf{n} := \mathsf{b_1}, \mathsf{c_2}] \setminus \{\!|\ k\ |\!\} \end{array}$$

The proof uses basic laws of CSP [5], which are also valid for ***Circus***. One of the laws is the distribution of injective renaming through parallel composition; this is expressed as $f(P \llbracket cs \rrbracket Q) = (f(P) \llbracket f(cs) \rrbracket f(Q))$ and referenced as $f[.]{-}\llbracket cs \rrbracket{-}dist$. Two other laws express the associativity $(P \llbracket cs \rrbracket (Q \llbracket cs \rrbracket R) = (P \llbracket cs \rrbracket Q) \llbracket cs \rrbracket R)$ and the symmetry $(P \llbracket cs \rrbracket Q = Q \llbracket cs \rrbracket P)$ of parallel composition. These are referenced bellow as $\llbracket cs \rrbracket{-}assoc$ and $\llbracket cs \rrbracket{-}sym$, respectively. Also, we use the fact that unused channels in the synchronism of a parallel operator are redundant, which is expressed as $P \llbracket \alpha P \cup \alpha Q \cup cs \rrbracket Q = P \llbracket \alpha P \cup \alpha Q \rrbracket Q$, where αR is the alphabet of R; we refer this law as $\llbracket cs \rrbracket{-}null$. Finally, we use the fact that the parallel operator is idempotent under certain restrictions [21]: if P is a deterministic process, then $P = P \llbracket \alpha P \rrbracket P$; this law is referenced as $\llbracket cs \rrbracket{-}idem$.

$$
\begin{aligned}
& Struct_{\mathsf{A}} \\
= & [1.\ \textit{Expanding } Chart_{\mathsf{A}} \textit{ in } Struct_{\mathsf{A}}] \\
& (Chart_{\mathsf{B}}[\mathsf{b_2} := \mathsf{k}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket\, Chart_{\mathsf{C}}[\mathsf{c_1} := \mathsf{k}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket \mathsf{X}[chan_{\mathsf{X}} := \mathsf{b_1}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket \\
& \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket\, \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket\, \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c_2}]) \setminus \{\!| k |\!\} \\
= & [2.\ \textit{Applying Law } \llbracket cs \rrbracket{-}idem \textit{ for } \mathsf{X}[chan_{\mathsf{X}} := \mathsf{b_1}] \textit{ and } \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c_2}], \textit{ and} \\
& \textit{rearranging the processes using Laws } \llbracket cs \rrbracket{-}assoc \textit{ and } \llbracket cs \rrbracket{-}sym] \\
& (Chart_{\mathsf{B}}[\mathsf{b_2} := \mathsf{k}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket \mathsf{X}[chan_{\mathsf{X}} := \mathsf{b_1}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket \\
& \mathsf{X}[chan_{\mathsf{X}} := \mathsf{b_1}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket\, Chart_{\mathsf{C}}[\mathsf{c_1} := \mathsf{k}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket\, \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket \\
& \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c_2}] \llbracket \{\!| \mathsf{k,b_1,c_2} |\!\} \rrbracket\, \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c_2}]) \setminus \{\!| k |\!\} \\
= & [3.\ \textit{Applying renamings } [\mathsf{b'_1,b_2} := \mathsf{m,k}],\ [\mathsf{c'_2,c_1} := \mathsf{n,k}] \textit{ and } [\mathsf{n,m} := \mathsf{c_2,b_1}], \textit{ and} \\
& \textit{Law } f[.]{-}\llbracket cs \rrbracket{-}dist] \\
& (((Chart_{\mathsf{B}} \llbracket \{\!| \mathsf{b_2,b'_1,n} |\!\} \rrbracket \mathsf{X}[chan_{\mathsf{X}} := \mathsf{b'_1}] \llbracket \{\!| \mathsf{b_2,b'_1,n} |\!\} \rrbracket \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{b_2}])[\mathsf{b'_1,b_2} := \mathsf{m,k}] \\
& \llbracket \{\!| \mathsf{k,m,n} |\!\} \rrbracket\, \mathsf{X}[chan_{\mathsf{X}} := \mathsf{m}]) \llbracket \{\!| \mathsf{k,m,n} |\!\} \rrbracket \\
& ((Chart_{\mathsf{C}} \llbracket \{\!| \mathsf{c_1,m,c'_2} |\!\} \rrbracket\, \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{c_1}] \llbracket \{\!| \mathsf{c_1,m,c'_2} |\!\} \rrbracket\, \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c'_2}])[\mathsf{c'_2,c_1} := \mathsf{n,k}] \\
& \llbracket \{\!| \mathsf{k,m,n} |\!\} \rrbracket\, \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{n}])[\mathsf{m,n} := \mathsf{b_1,c_2}]) \setminus \{\!| k |\!\} \\
= & [4.\ \textit{Using the Law } \llbracket cs \rrbracket{-}null \textit{ and rearranging the processes using laws}] \\
& \llbracket cs \rrbracket{-}assoc \textit{ and } \llbracket cs \rrbracket{-}sym \\
& (((Chart_{\mathsf{B}} \llbracket \{\!| \mathsf{b_2,b'_1} |\!\} \rrbracket \mathsf{X}[chan_{\mathsf{X}} := \mathsf{b'_1}] \llbracket \{\!| \mathsf{b_2,b'_1} |\!\} \rrbracket \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{b_2}])[\mathsf{b'_1,b_2} := \mathsf{m,k}] \\
& \llbracket \{\!| \mathsf{k,m,n} |\!\} \rrbracket\, ((Chart_{\mathsf{C}} \llbracket \{\!| \mathsf{c_1,c'_2} |\!\} \rrbracket \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{c_1}] \llbracket \{\!| \mathsf{c_1,c'_2} |\!\} \rrbracket \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c'_2}])[\mathsf{c'_2,c_1} := \mathsf{n,k}] \\
& \llbracket \{\!| \mathsf{k,m,n} |\!\} \rrbracket\, \mathsf{X}[chan_{\mathsf{X}} := \mathsf{m}]) \llbracket \{\!| \mathsf{k,m,n} |\!\} \rrbracket\, \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{n}])[\mathsf{m,n} := \mathsf{b_1,c_2}]) \setminus \{\!| k |\!\} \\
= & [5.\ \textit{Using the definition of } Struct_{\mathsf{B}} \textit{ and } Struct_{\mathsf{C}}] \\
& ((Struct_{\mathsf{B}}[\mathsf{b'_1,b_2} := \mathsf{m,k}] \llbracket \{\!| \mathsf{k,m,n} |\!\} \rrbracket\, (Struct_{\mathsf{C}}[\mathsf{c'_2,c_1} := \mathsf{n,k}] \llbracket \{\!| \mathsf{k,m,n} |\!\} \rrbracket \\
& \mathsf{X}[chan_{\mathsf{X}} := \mathsf{m}]) \llbracket \{\!| \mathsf{k,m,n} |\!\} \rrbracket\, \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{n}])[\mathsf{m,n} := \mathsf{b_1,c_2}]) \setminus \{\!| k |\!\} \\
= & Struct_{\mathsf{A'}}
\end{aligned}
$$

Three conditional CSP laws are used in the above proof. The condition of Law $\llbracket cs \rrbracket{-}idem$ (Step 2) is clearly satisfied because the processes X and Z are obtained from the protocols X and Z, which, in Law 1, are assumed to be deterministic. The condition of Law $\llbracket cs \rrbracket{-}null$ (Step 4) is satisfied since the processes $Chart_{\mathsf{B}}$, $\mathsf{X}[chan_{\mathsf{X}} := \mathsf{b'_1}]$ and $\mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{b_2}]$ do not use the channel n; similarly, processes $Chart_{\mathsf{C}}$, $\mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{c_1}]$ and $\mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c'_2}]$ do not use m. The condition of Law $f[.]{-}\llbracket cs \rrbracket{-}dist$ (Step 3) is satisfied since following renamings used in the distribution are injective: $[\mathsf{b'_1,b_2} := \mathsf{m,k}]$, $[\mathsf{c'_2,c_1} := \mathsf{n,k}]$ and $[\mathsf{n,m} := \mathsf{c_2,b_1}]$

6 Conclusions

We have proposed a formal semantics for capsules, protocols, ports and connections in UML-RT, using ***Circus*** as a semantic domain. We considered an

integrated view involving UML-RT class, state and structure diagrams. Also, we presented a transformation law for capsule decomposition that considers the effect on all these diagrams. Based on the semantic mapping, and on the semantics and laws of *Circus*, we have shown that it is relatively simple to prove such laws. As far as we are aware, an entirely formal approach to transformation of UML-RT models is an original contribution.

Although the target of our translation is a specification language (rather than a more standard mathematical model), the notation of *Circus* includes those of CSP and Z. Both are well-known and mature formalisms. Further, their combination into *Circus* is formally characterised with basis on the unifying theories of programming [20].

The basis for the translation given in this paper is a formalisation of the syntax of UML-RT in Z and in CSP. The formalisation of UML-RT structure diagrams is similar to [8], which formalises only this kind of diagram in CSP, and disregards other views and elements of the architecture, as statecharts and protocols. The work reported in [9] briefly presents some notions that could be used as a basis for a mapping from UML-RT into CSP, but based on these notions it seems rather difficult mapping complex systems in all its aspects without additional assumptions on the dynamic semantics. It briefly covers simple capsule statecharts, and does not give any additional contribution on the *capsule-connector-capsule* translation when compared to [8]. An informal translation of UML-RT to CSP-OZ is also reported in [7] through a case study. Despite the benefit of linking UML-RT to CSP-OZ and Java, the work in [7] does note seem concerned with the soundness of transformations for UML-RT, but rather with platform transformations between these languages. A more general contextualisation of our work has already been given in Section 1.

Currently, we are working on a comprehensive set of laws for UML-RT, exploring their use during the development of a more elaborate version of the example presented in Section 2. Our notion of completeness is based on showing that an arbitrary UML-RT model can be reduced to a UML model extended with a single capsule responsible for all the interactions with the environment. This extended UML model can be regarded as a normal form, and therefore relative completeness of our set of laws can be captured by normal form reduction, following an approach similar to [22].

In general, the laws involve classes and relationships between them as well as with capsules. The semantics of *OhCircus* [23] is being defined as a conservative extension of that of *Circus*, to address object-oriented features. We plan to extend our mapping to consider the full UML-RT notation, using *OhCircus* as semantic domain.

References

1. Object Management Group: OMG Unified Modeling Language Specification. (2003) OMG document formal/03-03-01.
2. Bruel, J.M., France, R., Larrondo-Petrie, M.: An integrated object-oriented and formal modeling environment. Journal of Object-Oriented Programming **10** (1997)

3. Ng, M.Y., Butler, M.J.: Towards Formalizing UML State Diagrams in CSP. In: 1st International Conference on Software Engineering and Formal Methods, IEEE Computer Society (2003)
4. Spivey, M.: The Z Notation: A Reference Manual. second edn. Prentice Hall (1992)
5. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall (1998)
6. Amalio, N., Polack, F.: Comparison of formalisation approaches of UML class constructs in Z and Object-Z. In: Third International Conference of B and Z. Volume 2651 of LNCS. (2003)
7. Möller, M., Olderog, E.R., Rasch, H., Wehrheim, H.: Linking CSP-OZ with UML and Java: A Case Study. In: 4th International Conference on Integrated Formal Methods. Volume 2999 of LNCS., Springer (2004)
8. Fischer, C., Olderog, E.R., Wehrheim, H.: A CSP View on UML-RT Structure Diagrams. In: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering, Springer (2001)
9. Engels, G., Küster, J.M., Heckel, R., Groenewegen, L.: A methodology for specifying and analyzing consistency of object-oriented behavioral models. In: Proceedings of the 8th European Software engineering Conference, ACM Press (2001)
10. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex RealTime Systems. Rational Software Corporation (1998) available at http://www. rational.com.
11. Engels, G., Heckel, R., Küster, J.M., Groenewegen, L.: Consistency-Preserving Model Evolution through Transformations. In: 5th International Conference on the Unified Modeling Language. Volume 2460 of LNCS., Springer (2002)
12. Sampaio, A., Woodcock, J., Cavalcanti, A.: Refinement in *Circus*. In: International Symposium of Formal Methods Europe. Volume 2391 of LNCS., Springer (2002)
13. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall (1998)
14. Fischer, C.: Combination and Implementation of Processes and Data: from CSP-OZ to Java. PhD thesis, Fachbereich Informatik Universität Oldenburg (2000)
15. Morgan, C.: Programming from Specifications. second edn. Prentice Hall (1994)
16. Sampaio, A., Mota, A., Ramos, R.: Class and Capsule Refinement in UML for Real Time. In: Proceedings of the Brazilian Workshop on Formal Methods. Volume 95 of ENTCS. (2004)
17. Object Management Group: OMG Adopted Specification. (2003) OMG documents ptc/03-09-15 and ptc/03-08-02.
18. Richters, M., Gogolla, M.: OCL: Syntax, Semantics, and Tools. In: Object Modeling with the OCL: The Rationale behind the Object Constraint Language. Springer (2002)
19. Selic, B.: An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems. In: 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications. Volume A-32 of IFIP Transactions. (1993)
20. Woodcock, J., Cavalcanti, A.: The Semantics of *Circus*. In: ZB 2002: Formal Specification and Development in Z and B. Volume 2272 of LNCS., Springer (2002)
21. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
22. Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic reasoning for object-oriented programming. Science of Computer Programming **52** (2004)
23. Cavalcanti, A., Sampaio, A., Woodcock, J.: A unified language of classes and processes. In: St Eve: State-Oriented vs. Event-Oriented Thinking in Requirements Analysis, Formal Specification and Software Engineering, Satellite Workshop at FM'03 (2003)