Transformation Laws for UML-RT

Rodrigo Ramos, Augusto Sampaio, and Alexandre Mota

Centre for Informatics, Federal University of Pernambuco
P.O.Box 7851, CEP 50740-540, Recife-PE, Brazil
{rtr, acas, acm}@cin.ufpe.br

Abstract. With model-driven development being on the verge of becoming an industrial standard, the need for systematic development strategies based on safe model transformations is a demand. Transformations must take into account changes in both behavioural and structural diagrams. In this paper, we present a set of transformation laws that aims to systematise the evolution of semantically well-defined UML-RT models, with preservation of both static and dynamic aspects. The proposed laws support the transformation of initial abstract analysis models into concrete design models. Furthermore, we show the seamless application of the laws through design activities of the Rational Unified Process in the development of a case study. Soundness and completeness of the laws are briefly addressed.

1 Introduction

In Model Driven software Engineering (MDE) [1], the central artifacts, and the driving force, of a software development are models, rather than code in a programming language. As a departure from the general idea that the usefulness of models are only for documentation or to capture interesting design aspects during development, the main objective in MDE is that the development process is driven by the activity of modelling.

The purpose of MDE is combining an architecture of the model roles with other process activities. In this framework, transformations help to overcome the challenges of model evolution, allowing restructure of the software with preservation of behaviour; the idea is similar to well-known code transformation techniques like refactorings [2] and refinements [3].

Models are widely expressed using the Unified Modeling Language (UML) [4], as well as its extensions. In particular, we emphasise the use of UML-RT [5], an UML profile, which has a clear definition for reactive components and component protocols, and is useful to describe concurrent and distributed domains. In previous work [6] we defined a formal semantics for UML-RT, and illustrated how model transformations can be verified. We have also explored refinement notions for UML-RT, together with some large grain transformation rules [7], which seemed useful to support the evolution and restructure of architectural UML-RT models.

Here we concentrate on an algebraic presentation of UML-RT. We propose a set of algebraic laws for the language, focusing on the new elements that it adds to UML: active classes (capsules), protocols, ports and connections. The proposed laws express both basic properties of each individual construct, as well as relationships among them. For instance, the laws permit justifying transformations of initial abstract models into concrete design models, using consistent steps with preservation of both static and dynamic model aspects (taking into account structural and behavioural properties together); this is illustrated through the development of a case study: a simple manufacturing system. We also address a notion of completeness, briefly presenting a strategy to reduce an arbitrary UML-RT model to a extended UML model, entirely based on the laws.

Our work can be considered complementary to others that focus on laws for UML design elements [8, 9], as well as on component transformations [10]. A more detailed account of related work is left for the concluding section.

We briefly present an overview of UML-RT, including part of a formal semantics, in the next section. In Section 3, we present a selection of our set of laws, and address completeness. In Section 4, we show how these laws can be used to capture and formalise some design guidelines adopted, for example, by the Rational Unified Process (RUP) [11], through the development of our case study. Our conclusions and related work are presented in Section 5.

2 UML-RT Semantics Overview

The specification of reactive systems is a complex task, involving data, control behaviour, communication and architectural modelling aspects. In order to incorporate support for all these facets into a widely used language, like UML, several ROOM concepts have been added to the UML-RT. Although some of these concepts have also influenced the component model of the recent UML 2.0 [12] version, here we use the UML-RT profile because we consider that its proposed model for active objects is more consolidated than that proposed for UML 2.0. A detailed comparison between UML-RT and UML 2.0 is out of the scope of this paper; the work reported in [13] presents some problems on the statechart specification of UML 2.0. Furthermore, there is commercially established tool support for UML-RT.

UML-RT, like other Architectural Description Languages, models reactive systems with active architectural components working concurrently and communicating among themselves. Communication is modelled by means of input and output message exchanges, which can be synchronous or asynchronous; here we assume synchronous communication. These concepts have been introduced to UML-RT via four new design elements: capsule, protocol, port and connector. Capsules (active classes) describe architectural components whose points of interaction are called ports, which are assembled by connectors and realise communication signals previously declared in a protocol.

Despite the expressiveness of UML-RT, the rigorous development of non-trivial applications does not seem feasible without an assigned formal semantics. In previous work [6] we defined a semantics for UML-RT via mapping into the formal notation ***Circus*** [14], which combines CSP, Z and specification statements.

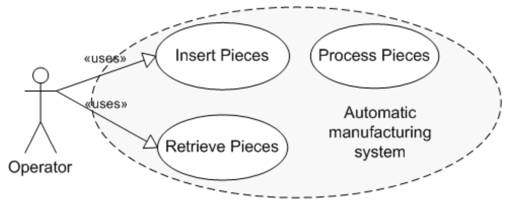


Fig. 1. Use Case Diagram

The semantics of *Circus* is defined in the setting of the Unifying Theories of Programming [15]; this relational model has proved convenient for reasoning. Another advantage is that *Circus* includes the main design concepts of UML-RT and provides a refinement calculus [3]. Both the semantics and the laws of *Circus* have been inspiring to prove laws that we propose for UML-RT.

Throughout this section, an example of a simplified manufacturing system is used to illustrate the UML-RT notation and semantics. In this system, the entire application is responsible for processing a number of workpieces, which are inserted by an operator and made available for retrieving after processed. These functionalities are presented by the use case diagram in Figure 1. Processing pieces is an autonomous process that does not require any operator intervention.

In Figure 2, an abstract model of this system is presented. The model is formed by a set of diagrams and system properties, using diagrams that mainly represent the following architectural views: static data, dynamic behaviour, and instance 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, but an OCL to *Circus* mapping is out of the scope of this work.

In the class diagram (top left rectangle) of Figure 2, capsules and protocols are graphically represented by stereotyped classes with labels Capsule and Protocol. The diagram emphasises the relationships between the capsules ProdSys and Storage. The capsule Storage is a bounded reactive 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 STO and STI, which are used to govern their communications: STO declares the input signal req and the output signal output (used to communicate the request and the delivery of a work piece, respectively), while 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 only visible elements in the capsule are ports, which can be connected to other capsule ports to establish communication. This decoupling makes capsules highly reusable. In addition, a capsule can also be defined hierarchically.

A structure diagram describes a capsule structural decomposition in sub-capsules, showing the capsule interaction through connections among its ports. We assume that a configuration of the manufacturing system, given by the

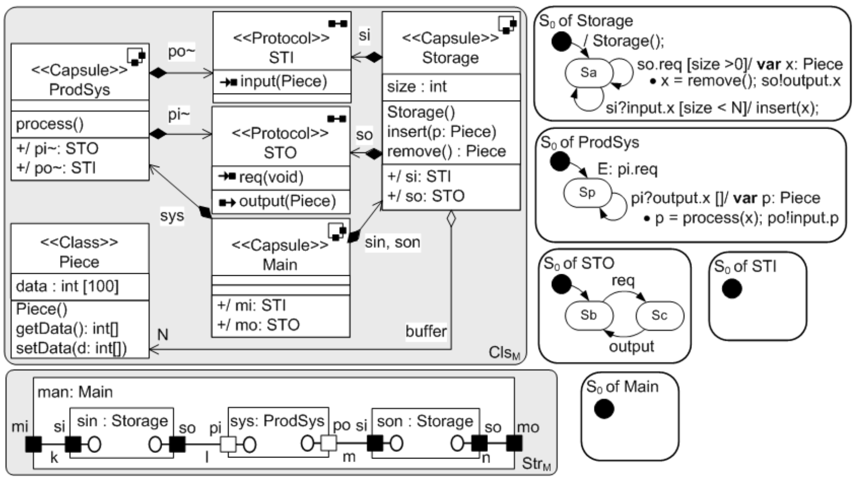


Fig. 2. Abstract Analysis Model

structure diagram in Figure 2 (bottom rectangle), is represented by an instance (man) of capsule Main, which is structurally decomposed into the sub-capsules sin, son and sys; these sub-capsule instances are created as a consequence of the association of Main with Storage and ProdSys in the class diagram. Black filled squares in the capsule instances represent their ports, which are used for communication (*end ports*) or just to convey signals to other sub-capsules (*relay ports*). Each end port can be connected only to another *conjugated* port of the same protocol; conjugated ports are represented by (unfilled) squares and have the directions of their input and output signals inverted in order to fully assemble with other ordinary ports. For instance, in the structure diagram, the ports pi and po are public end ports of ProdSys, while mi and mo are public relay ports of Main used only to connect ports of sub-capsules to the environment.

The capsule behaviour is described in terms of UML-RT statecharts, which differ from the standard UML statecharts [16] by including some adaptations to better describe active objects. 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.

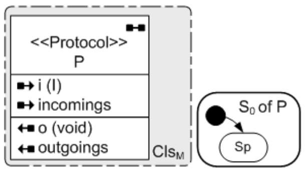
We assume that events, guards and actions are expressed using the *Circus* notation. For example, in the statechart of Storage, there are two transitions from state Sa. The one on the right triggers if the req signal arrives through port so and the buffer is non-empty. The corresponding action declares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

The formal counterpart of the UML-RT concepts are also found in *Circus*, since, in this language, concurrent components are represented by processes that interact via channels. Therefore, capsules and protocols, classifiers with an associated behaviour, are semantically mapped into processes, ports into channels, and classes into Z paragraphs, which act as passive data registers. Furthermore, connections are represented by means of using common channels. As an example, consider the following mapping of the capsule Storage into *Circus*.

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

In *Circus*, a process declaration body (delimited by the **begin** and **end** keywords) is composed of a Z state schema, action paragraphs and a main action (delimited after the • symbol), which defines the process behaviour; action paragraphs are used to structure the behaviour of a main action and to express data operations. In this specification, the maximum size of the buffer is a positive constant N. The Storage process 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 our example, the process Storage encapsulates two state components in the Z schema *StorageState*: an ordered list buffer of contents and the size of this list. Initially, buffer is empty and, therefore, its size is zero; this is specified as a state initialisation action *StorageInit*. Pre- and postconditions of methods, and state invariants play a corresponding role as annotations in the model; they have not been included in Figure 2 for conciseness. The main action initialises the buffer and then acts like the action Sa, repeatedly offering the choice of input and req, like Sa in the statechart of Storage (Figure 2). The main action represents the topmost state (S_0) of Storage; this contains all other states in the statechart, and each of these enclosing states is also mapped to another action paragraph. In the following we present a semantic mapping for protocols and capsules.

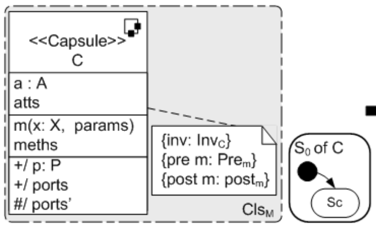
A protocol declaration in UML-RT encapsulates both the communication elements (signals) and the allowed flow of these elements (statechart). In *Circus*, this gives rise to two major elements: a process that captures this behaviour and a channel to represent the communication elements. This single channel communicates values of a free type, with each constructor representing a signal.



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

In names like $chan_{\mathsf{P}}$ above, we assume that P is a placeholder for the actual protocol name. The channel $chan_{\mathsf{P}}$ communicates values of the free type T_{P}; each value represents a signal. Parameterless and parameterised signals are mapped into constants and data constructors, like the signals o and i above. The type of the parameter is translated into a corresponding *Circus* type by the function $\mathcal{T}()$. The remaining signals (incomings and outgoings) are mapped by the meta function $\mathcal{TL}()$ that translates this remaining lists such as the elements that was singled out. The main action is represented by $\mathcal{H}(\mathsf{S_P})$, where $\mathsf{S_P}$ stands for the topmost composite state of P and the function $\mathcal{H}()$ translates a statechart into a *Circus* action.

Capsules are also defined as processes, with methods and attributes defined as Z operation and state schemas. 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 protocol signals.



channel $\mathsf{p} : T_{\mathsf{P}};\ \mathcal{TL}(\mathsf{ports});\ \mathcal{TL}(\mathsf{ports'})$
process $Chart_{\mathsf{C}} \mathrel{\widehat=}$ **begin**
  state $\mathsf{C}_{state} \mathrel{\widehat=} [\mathsf{a} : \mathcal{T}(\mathsf{A});\ \mathcal{TL}(\mathsf{atts}) \mid \mathsf{Inv_C}]$
  $m \mathrel{\widehat=} [\Delta \mathsf{C}_{state};\ \mathsf{x} : \mathcal{T}(\mathsf{X});$
    $\mathcal{TL}(\mathsf{params}) \mid \mathsf{Pre_m} \land \mathsf{Post_m}]$
  $\mathcal{TL}(\mathsf{meths})$
  $\bullet\ \mathcal{H}(\mathsf{S_c})$
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, and remaining lists in these compartments are mapped by the function $\mathcal{TL}()$. 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 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. The invariant $\mathsf{Inv_C}$, preconditions $\mathsf{Pre_m}$ and postconditions $\mathsf{Post_m}$ come from the UML-RT note element on the left, and it is assumed to be already described 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.

In *Circus* the semantics of the dynamic behaviour of a capsule C, including its internal structure diagram, is captured by the parallelism of its internal behaviour ($Chart_C$), its connected sub-capsules and its ports. The dynamic behaviour considers restrictions imposed by its ports to the corresponding communication channels and the interaction with its sub-capsules in a hierarchical and compositional way. This is expressed in *Circus* by means of a parallel composition of these processes. Further details about our semantics for UML-RT is presented in [6].

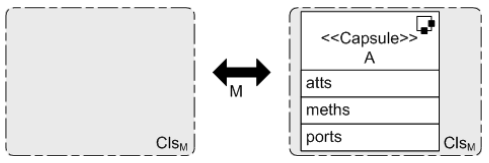
3 Transformation Laws

Based on the formal semantics briefly presented in the previous section, we propose some transformation laws for UML-RT. As with the formal semantics, we concentrate on the elements that UML-RT adds to UML, and the relationships of these with UML elements, especially classes. The laws deal with static and dynamic model aspects represented by the three most important diagrams of UML-RT: statechart, class and structure diagrams. The proof of the laws can be found in [17, 6], using both semantics and the refinement laws of *Circus* [14].

Each law is defined by an equivalence relation on models (filled arrow) with a subscript M that stands for the context in which the equality holds; the soundness of this equivalence relation is based on the formal semantics: the models on the two sides of each law are semantically equivalent, when mapped into *Circus* specifications. Our laws do not modify the context M, but impose side conditions on some of its views: $\mathsf{Cls_M}$ represents the class diagrams of M, and $\mathsf{Str_M}$ denotes the architecture configuration of M, expressed by structure diagrams. State diagrams of protocols and capsules are explicitly expressed in the law. Diagrams and notes describing properties (invariants, pre- and postconditions) are presented only as the need arises. On each side of the law we use a dotted line box to single out the relevant part of the model affected by the transformation.

The first law establishes when it is possible to introduce a new capsule into the model.

Law 1. *Declare Capsule*



provided

(→) $\mathsf{Cls_M}$ *does not declare any element named* A.

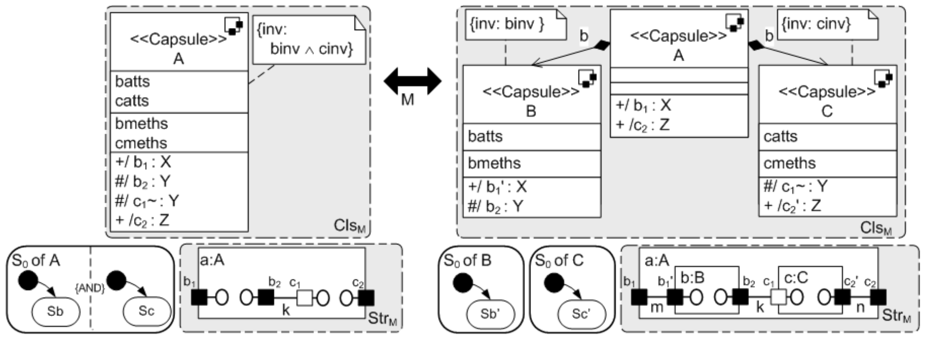
(←) *No capsule in* M *has a relationship with capsule* A *in any diagram.*

The left-hand side of Law 1 displays an empty box, meaning that the new capsule can be included anywhere in the model, provided the side conditions are satisfied; the subscript M fixes the context for the law application. We write (→) before the proviso of a law to indicate that it is required only for applications of this law from left to right. Similarly, we use (←) to indicate that it is only for applying the law from right to left, and we use (↔) to indicate that the proviso is necessary in both directions.

A proviso to remove a capsule A (application from right to left) is that no other capsule extends, is associated to or connected with it. Since UML (and UML-RT) does not allow two elements with the same name, then there is a proviso stating that the name of the new element is fresh. We assume that atts, meths and ports always represent the set of attributes, methods and ports of a capsule. As the capsule is assumed not to be associated with any other in M, the structure diagram or a statechart can have any form that obeys our provisos, and their presentation is immaterial. We have similar laws to add or remove other basic elements (for instance, protocols, ports and connections).

The next law captures a more elaborate transformation; it 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 be partitioned, a concept that is explained next. Note that protocols X and Z are not illustrated because any deterministic machines can be used.

Law 2. *Capsule Decomposition*



provided

(→) $\langle$batts, binv, bmeths, (b_1, b_2), Sb$\rangle$ *and* $\langle$catts, cinv, cmeths, (c_1, c_2), Sc$\rangle$ *partition* A.
(↔) *The statecharts of the protocols* X *and* Z *are deterministic.*

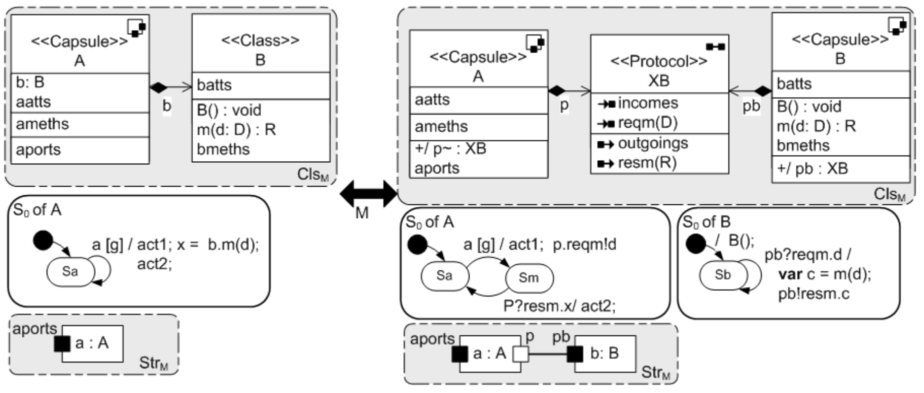
On the left-hand side of Law 2 the state machine of A is an And-State composed of two states (Sb and Sc), which may interact (internal communication) through the conjugated ports b_2 and c_1 (as captured by the structure diagram on the left-hand side). The two other ports (b_1 and c_2) are used for external 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 binv ∧ cinv, where binv involves only batts as free variables, and cinv only catts. When a capsule obeys such conditions, we say that it is partitioned. In this case, there are two partitions: one is $\langle$batts, binv, bmeths, (b_1, b_2), Sb$\rangle$ and the other is $\langle$catts, cinv, cmeths, (c_1, c_2), Sc$\rangle$.

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 2 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.

Motivated by existing development practices, we propose a law that replaces a capsule by a class, or vice-versa. This establishes an interesting connection between passive and active classes.

Law 3. *Replace a Class by a Capsule*



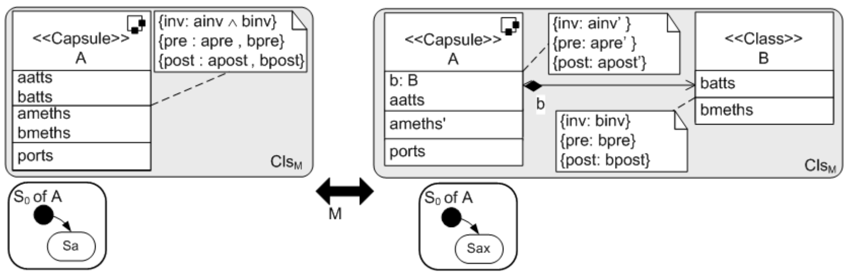
provided

(→) *All attributes within* batts *are private.*
(↔) *No capsule, except* A*, has a relationship with* B.

Law 3 transforms a class B (left-side) into a capsule, also named B, on the right-hand side. The behaviour of method calls is preserved by a statechart that simulates a synchronised communication with the client protocol. Now, all services (public methods) of the class are exposed in a new protocol XB. The constructor of class B becomes an action in its statechart's initial transition.

The following law promotes a hidden abstraction inside a capsule into an independent passive class. This simple transformation seems recurrent during several design steps. It is particularly helpful in the context where the system is initially modelled as components, since it allows the extraction of relevant classes from these components. Concerning side conditions, we need to consider the effect on the state machine of the capsule, whose behaviour must be preserved by the transformations. A simplified version of this law allows extracting a class from another class.

Law 4. *Extract Class*



provided

($\rightarrow$) bmeths, binv, bpre *and* bpost *refer only to methods and attributes within* bmeths *and* batts; B *is a fresh identifier.*

($\leftarrow$) *No element, except* A*, refers to* B*; there is an equivalence relationship between* Sax *and* Sa*, and also between* ameths *and* ameths′.

On the left-hand side of law 4, capsule A is composed by the set of attributes aatts and batts, of methods ameths and bmeths, and of ports ports. Its state machine, represented by Sa, can access any of these elements. Its invariant is represented by the conjunction ainv $\wedge$ binv, and the pre- and postconditions of its methods are captured by apre, bpre, apost and bpost. The elements inside batts, bmeths, bpre and bpost are assumed not to refer to any other element of A. These are the elements that will be extracted into a new class.

On the right-hand side, any action in Sax, methods within ameths′ or predicates (ainv′, apre′ or apost′) of A, which on the left-hand side accessed attributes of batts or methods of bmeths, will now access these elements via a qualifier b, which represents an object of the new class B.

3.1 A Word on Completeness

One way of showing that a set of laws is comprehensive is to define a reduction strategy based on the laws, whose target is a normal form described in terms of a restricted subset of the language being discussed. This shows that the laws are sufficiently powerful to reduce any program to this normal form, and moreover any pair of equivalent programs to the same normal form. This is what we briefly discuss here, for a normal form that extends a UML model with a single capsule responsible for all the interactions with the environment; this capsule also centralises the entire active behaviour of the modelled system. The reason for keeping one active element as part of the normal form (rather than a pure UML model containing only passive classes) is that the autonomous control flows of the original capsules cannot be simply eliminated; the closest we can get is combining them as a single statechart. Reducing an arbitrary UML-RT model to such a form suggests that our laws are expressive enough to reason about the new design elements that UML-RT adds to UML, which is our major concern.

As it is not possible to present all the equality laws used by our structural reduction strategy, we list some categories of laws below; the laws already presented fall in these categories. We also refer to these categories in the development of our case study.

– *Laws of declaration*: for introducing/removing capsules (Law 1), protocols, ports, signals of a protocol, attributes, methods, and associations.
– *Communication*: for introducing new connections and intermediate capsules.
– *Merging*: combining/decomposing capsules (Law 2), protocols and ports.
– *Delegation*: transferring part of a protocol/capsule behaviour to another capsule, protocol or class (for instance, Law 4).
– *Structuring*: Encapsulate (making it local) or replace a capsule with another capsule or class (for instance Law 3)
– *Statechart*: Adding a new state, rewriting a transition action, partitioning regions, among others.

Laws in these categories are used in the main steps of our reduction strategy, as summarised below.

1. Merge capsule ports. Capsules should have a unique binary connection between them; this is justified by *Laws of Merging.*
2. Eliminate capsule hierarchy in structure diagrams. In this step, sub-capsules are moved out the structure of its enclosing capsule; this is justified by *Laws of Structuring.*
3. Move protocol behaviour to capsules. The entire system behaviour is expressed in its capsules; this is justified by the *Laws of Delegation.*
4. Capsule composition. Every two capsules that communicate should be encapsulated in a topmost capsule, and then composed; this is justified by *Laws of Merging.*
5. Remove unreferenced model elements. All disconnected elements can be removed from the model by the application of *Laws of Declaration.*

As a result of this strategy all the active elements are transformed into a single capsule that centralises the entire autonomous behaviour of the model. The (passive) classes are not affected by the strategy. This model might be reduced even further, by eliminating the classes as well; Law 4 (in its reverse order) is relevant here. However, for a complete elimination of classes, possibly involving inheritance, additional laws would be necessary, as suggested in [18] for a programming language; our focus here is on active rather than on passive classes. Our complete set of laws and more details of the reduction strategy can be found in [17].

Clearly, the objective of such a reduction strategy is merely to study the expressiveness of a set of laws. In a development process, the laws are applied in the reverse direction, supporting the evolution of simple and abstract models into more elaborate design models, as illustrated in the next section.

4 Case Study

The transformation laws we have proposed may be useful to formalise informal analysis and design guidelines widely adopted by development processes such as, for instance, the Rational Unified Process (RUP) [11]. The analysis and design disciplines of RUP include several activities that guide the developer to systematically realise the use case view into the so-called logical view. Broadly, abstractions are identified, an abstract (analysis) model is developed, and this is progressively refined into a concrete design model.

The focus of our approach is to support a formal transition from analysis into design. Complementary approaches, like [19], for example, address the rigorous migration from the use case view to an initial abstract model (Figure 1) with active objects in the logical view. We allow great flexibility concerning the starting point for our development. An extreme could be a centralised model with a single capsule, just like the normal form discussed in the previous section. In the context of our case study, such a model would include a single capsule Main with all the interactions described in the use case diagram in Figure 1.

The proposed laws can then be applied to evolve this monolithic model into a concrete detailed model like the one in Figure 2. For example, Law 2 justifies the decomposition of Main into ProdSys and Storage, and Law 4 justifies the extraction of class Piece from Storage. The remainder of this section further refines this model to a more concrete version, with more than one processor, working in a pipeline, and a transportation agent (Holon) that intermediates the communication.

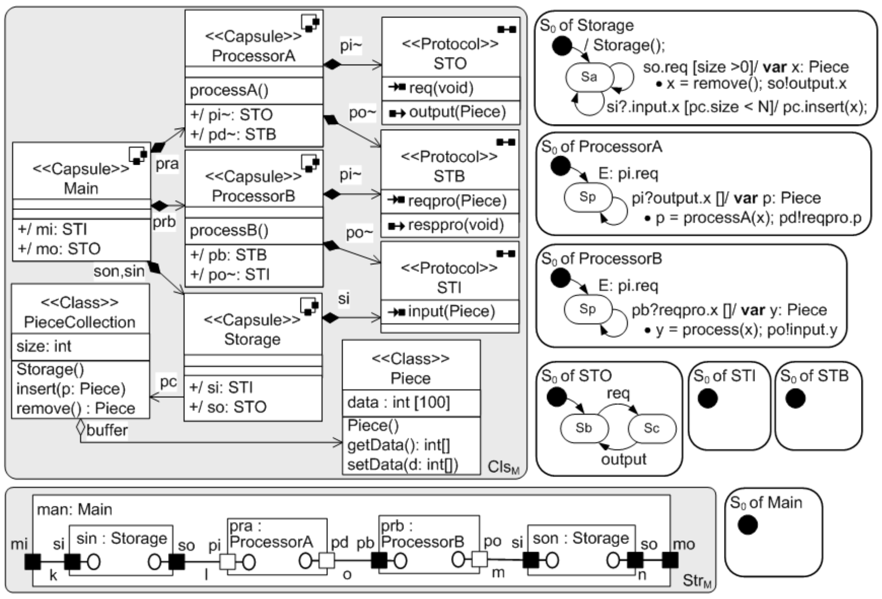


Fig. 3. Extracting PieceCollection and Decomposing the Processor

From the analysis model in Figure 2, we proceed to find a *candidate architecture*. We adopt a simple layered architecture where data manipulation is isolated from the business rules of the control elements. Therefore we use an explicit data collection class PieceCollection to store the workpieces; this class is actually extracted from Storage using Law 4.

The candidate architecture is incrementally enhanced by means of the activity *identify design elements*. In particular, we decompose the processor ProdSys into two other capsules (ProcessorA and ProcessorB), using Law 2 as well as *Laws of Statecharts*, *Communication* and *Delegation*. With the introduction of ProcessorA and ProcessorB, the statechart of ProdSys is split into the statecharts of these new processors, and it is then represented by the interaction between ProcessorA and ProcessorB. The remaining role of ProdSys is only to mediate communication with these processors through its delay ports, and it is transferred to Main (*Laws of Communication*); then ProdSys becomes useless and can be eliminated (Law 1). The resulting model is presented in Figure 3.

Transportation agents are needed not only to intermediate the communication between physically separated processors, but also to relieve processors from concerns of the global processing plan. To create one transportation agent (Holon), we introduce intermediate capsule instances among the capsules that communicate with the processor (using *Laws of Communication*); these capsules need to be composed pairwise using a variation of Law 2 (*Laws of Merging*). As Holon

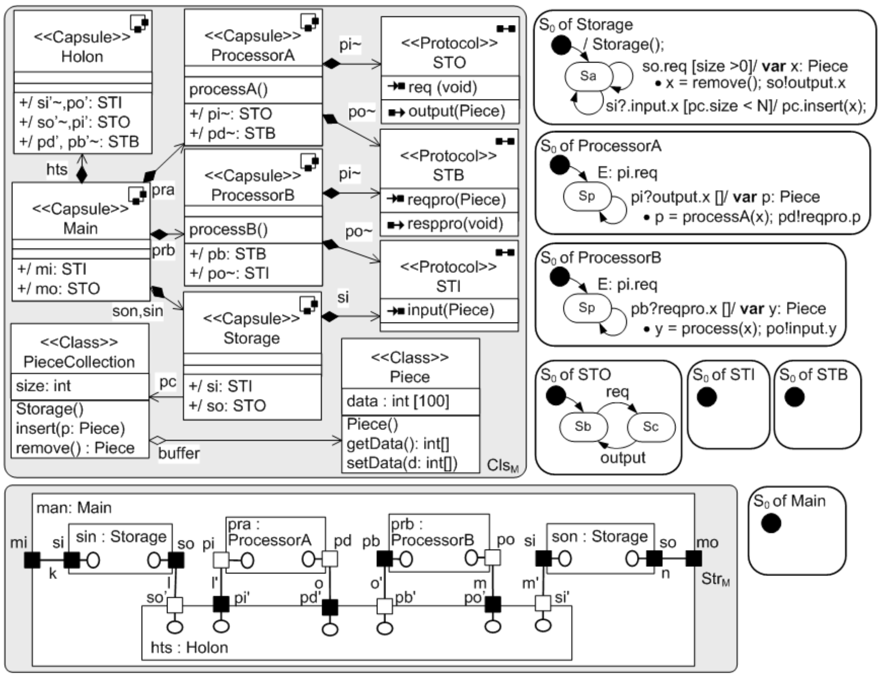


Fig. 4. Identifying Transportation Agents

was created by the composition of proxy capsules, it plays only a delegating role at this moment. The resulting design is depicted in Figure 4.

The result of the entire development is a system with a structure very close to the final architecture described in [20], but with a unique transport agent. A further refinement could extend the design to include some new agents to form a more complex automated transportation system, as well as refining the behaviour and the structure of each agent to a more concrete version. An advantage of our development strategy when contrasted to [20] is the justification of each design decision using transformation laws; no algebraic law has been proposed in [20].

5 Conclusions and Related Works

We have proposed laws for UML-RT that capture both basic properties of individual design elements as well as more elaborate transformations that correspond, for instance, to refactorings [2, 21]. The presentation of the laws makes explicit the transformation effects both on behavioural and on structural diagrams. Considering the elements that UML-RT adds to UML, our set of laws is comprehensive, as discussed in Section 3.1, and can be regarded as an algebraic semantics of these design elements. Another important issue is the connection between classes and capsules, as captured by Law 3. We have also suggested a guide for the law applications based on the RUP analysis and design discipline, as illustrated through the case study developed in Section 4. Soundness has been previously addressed through mapping into ***Circus*** [6], which acts as a hidden formalism, useful to define a sound interface for software engineering practice. The proof of the laws presented here, as well as the complete set of laws, can be found in [17].

Regarding laws for UML models, there are several works [21, 8, 9] that consider only transformations on structural or on behavioural diagrams in isolation. They neglect possible interferences between static and dynamic aspects, unlike our approach that takes into account these effects simultaneously. In [21], an important relationship between code refactoring concepts and model transformations is presented. Other approaches [8, 22, 9] define a formal semantics for UML using, respectively, Z, Alloy and Real-time Action Logic.

Regarding transformations for UML-RT, the work reported in [23] discusses a stepwise development process using UML-RT, incorporating notions of refinement, based on principles of *behavioural interface refinement* and *incorporating time*. In [24] the *locality principle* is explored, formalising model evolution using some local transformations; it also analyses the effects of these transformations on various consistency properties. None of these works, however, makes side conditions of laws explicit, presents a comprehensive set of laws for UML-RT, or systematises a strategy for algebraic-based model transformations, as we have done here. The work [10] also proposes an elaborated set of refactorings similar to ours, but the semantics used does not allow usage of an algebraic strategy.

As future work we intend to investigate the formalisation of design and architectural patterns based on our laws, and build tool support for automated transformations related to component based development.

References

1. Kent, S.: Model driven engineering. In: Proc. of the IFM Conference. Volume 2335 of LNCS., Springer (2002) 286–298
2. Fowler, M.: Refactoring-Improving the Design of Existing Code. Addison Wesley (1999)
3. Morgan, C.: Programming From Specifications. second edn. Prentice Hall (1994)
4. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language User Guide. Addison-Wesley (1999)
5. Selic, B., Rumbaugh, J.: Using UML For Modeling Complex RealTime Systems. Rational Software Corporation (1998) available at http://www. rational.com.
6. Ramos, R., Sampaio, A., Mota, A.: A Semantics for UML-RT Active Classes via Mapping into *Circus*. In: Proc. of the FMOODS Conference. Volume 3535 of LNCS., Springer (2005) 99–114
7. Sampaio, A., Mota, A., Ramos, R.: Class and Capsule Refinement in UML For Real Time. In: Proc. WMF'03. Volume 95 of ENTCS., Elsevier (2004) 23–51
8. Evans, A., France, R., Lano, K., Rumpe, B.: The UML as a Formal Modeling Notation. In: Proc. of the UML Conference. LNCS, Springer (1999)
9. Lano, K., Bicarregui, J.: Semantics and Transformations For UML Models. In: Proc. of the UML'99. Volume 1618 of LNCS., Springer (1999) 107–119
10. Meng, S., B.L., Naixiao, Z.: On refinement of software architectures. In: Proc. of the ICTAC Conference. Volume 3722 of LNCS., Springer (2005) 482–497
11. Kruchten, P.: Rational Unified Process: An Introduction, The. 2 edn. Addison-Wesley (2000)
12. OMG: UML 2.0 Superstructure Specification (2003) OMG Adopted Specification.
13. Fecher, H., Schönborn, J., Kyas, M., de Roever, W.P.: 29 New Unclarities in the Semantics of UML 2.0 State Machines. In: Proc. of the ICFEM Conference. Volume 3785 of LNCS., Springer (2005) 52–65
14. Sampaio, A., Woodcock, J., Cavalcanti, A.: Refinement in *Circus*. In: Proc. of the FME Symposium. Volume 2391 of LNCS., Springer (2002) 451–470
15. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
16. OMG: Unified Modeling Language Specification, Version 1.4. Object Management Group. (2001) Available at http://www.omg.org/uml.
17. Ramos, R.: Desenvolvimento Rigoroso com UML-RT. Master's thesis, Federal University of Pernambuco, Recife, Brazil (2005)
18. Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic Reasoning for Object-Oriented Programming. Science of Computer Programming **52** (2004)
19. Zhang, L., Xie, D., Zou, W.: Viewing Use Cases As Active Objects. ACM SIGSOFT Software Engineering Notes **26** (2001) 44–48
20. Wehrheim, H.: Specification of an Automatic Manufacturing System: A Case Study in Using Integrated Formal Methods. In: Proc. of the FASE Conference. Volume 1783 of LNCS., Springer (2000) 334–348
21. Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.M.: Refactoring UML Models. In: Proc. of the UML'01. Volume 2185 of LNCS., Springer (2001) 134–148
22. Gheyi, R., Borba, P.: Refactoring Alloy Specifications. In: Proc. WMF'03. Volume 95 of ENTCS., Elsevier (2004) 227–243
23. Sandner, R.: Developing Distributed Systems Step By Step With UML-RT. In: Proc. of the VVVNS Workshop, Universität Münster (2000)
24. Engels, G., Heckel, R., Küster, J.M., Groenewegen, L.: Consistency-Preserving Model Evolution Through Transformations. In: Proc. of the UML Conference. Volume 2460 of LNCS., Springer (2002) 212–226