Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 95 (2004) 23–51

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Class and Capsule Refinement in UML for Real Time

Augusto Sampaio [1] Alexandre Mota [2] Rodrigo Ramos [3]

Centro de Informática, Universidade Federal de Pernambuco, P.O.Box 7851
CEP 50740-540, Recife-PE, Brazil

Abstract

We propose refinement laws for the top level design elements of Real Time UML (UML-RT): classes and capsules. These laws can be used to develop concrete design models from abstract analysis models. Laws for introducing and decomposing classes and capsules are presented. Standard data refinement techniques are adapted for classes, and process refinement techniques for capsules. We also propose techniques for behavioural inheritance of classes and capsules. Soundness is briefly addressed by relating UML-RT elements to *OhCircus*, a formal unified language of classes and processes. To illustrate the overall strategy, we develop a detailed design of an operating system resource scheduler from a high-level analysis model.

Keywords: Class refinement, process refinement, behavioral inheritance, laws for UML-RT

1 Introduction

The integration of formal languages with informal (or semi-formal) notations has been a great challenge to the software engineering community, as this brings the much expected hope that the theories, techniques and methods carefully conceived in academia would play a decisive role in industrial software development practices.

Several approaches have been proposed. Some combine formal and informal languages into a single notation, in a complementary way. OCL [28] itself

[1] Email: acas@cin.ufpe.br
[2] Email: acm@cin.ufpe.br
[3] Email: rtr@cin.ufpe.br

1571-0661/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2004.04.004

is an example of this approach; it is used jointly with UML [2], with the purpose to annotate UML models, rather than giving a semantics to the UML constructs.

An alternative to combine notations is to assign meaning to an informal language by mapping into a formal notation (or semantic model). In this category we can find several proposals in the literature. In [14] the specification language TROLL is used to formalise UML in such a way that both formalisms can be applied for modelling. In [11] a simplified version of UML-RT (the Unified Modelling Language for Real Time) [7] is mapped into CSP [18,29], with constraints expressed in Object-Z [31], rather than in OCL.

This latter approach allows an interesting separation of concerns: the original notation continues to be used as before but, due to the existence of a formal semantics, techniques and methods proved sound for the formal language can be adapted to the informal notation. In this context the formal notation acts as a hidden formal model, useful to define a sound interface for software engineering practice.

Motivated by this research direction, the aim of this paper is to define a refinement strategy for UML-RT. The proposed strategy deals with *classes* and *capsules*. A capsule models an active class in the sense that, in addition to attributes and methods, its dynamic behaviour is given by a *state machine*. The interaction of a capsule with its environment (other capsules) is achieved via *protocols*. Typically, the only visible elements of a capsule are the signals introduced in the protocols to which a capsule is attached. Despite its name, the focus of UML-RT is on modelling concurrent and distributed aspects through the concept of a capsule, rather than on time aspects. Time concerns are out of the scope of this paper and is one of the topics for further research.

Like UML, UML-RT has no formal semantics, and therefore no proved sound refinement strategy. Our approach is to define refinement laws and techniques, for classes and capsules, based on results already consolidated in formal approaches. In particular, we briefly address soundness of the proposed laws and techniques by suggesting a mapping into the formal notation *OhCircus* [4], a unified language of classes and processes. While the work reported in [11] also addresses a mapping into a formal notation, refinement is not considered.

The language *OhCircus* extends *Circus* [34,5] with object-oriented features (classes, inheritance and dynamic binding). The other elements of *OhCircus* are exactly as in *Circus*, including the notion of a process, whose state is defined using a Z [35] schema and whose behaviour is given by an action expressed in the CSP notation. The semantics of *Circus* [34] (and the one currently being defined for *OhCircus*) is based on the unifying theories of programming [19]. A refinement strategy for *Circus*, which has been the main source of inspiration for this work, is reported in [30,5]. Both *Circus* and *OhCircus* have a *copy* semantics. Adopting *OhCircus* as our semantic model, we propose transformation laws which do not involve *sharing* (references). Dealing with references at the semantic and at the UML-RT levels is a future research topic.

One of our purposes is to preserve the style of UML-RT as much as possible. However, following a contractual approach [25], the characterisation of the laws needs to refer to invariants of classes and capsules, as well as to pre- and postconditions of methods. The standard notation used to annotate the model with formal constraints is OCL, but we use Z for a matter of readability and for simplifying the mapping into *OhCircus*.

The next section introduces UML-RT through a simple example: an abstract analysis model of a simplified operating system. Section 3 presents refinement laws and techniques for capsules and classes. The application of the proposed laws are illustrated by the development of a concrete design model for the resource scheduler component of the operating system, which is the purpose of Section 4. Soundness of the laws is briefly discussed in Section 5, where we present initial ideas towards a mapping from UML-RT into *OhCircus*. The final section describes a summary of our results and topics for further research.

2 UML-RT

UML-RT uses the basic UML mechanisms of stereotypes and tagged values for defining three new constructs: *capsule*, *protocol* and *connector*. For each construct, a corresponding stereotype is introduced: `<<capsule>>` and `<<protocol>>` stereotype classes, whereas `<<connector>>` stereotypes associations.

Capsules describe possibly complex active classes that may interact with their environment through messages (input/output signals). To each capsule is associated a unique behavior, given by a state machine. In addition, a capsule can also be defined hierarchically, in terms of compound capsules, each of which with a state machine and possibly a hierarchy of further compound capsules.

Protocols define the only possible way a capsule can interact with its environment, offering a set of input/output signals, like services of an interface. The actual interaction occurs through *ports*, which are declared in the respective capsules. Ports are instances of *protocols* and can regulate the flow of information (the protocol might have its own state machine to describe this

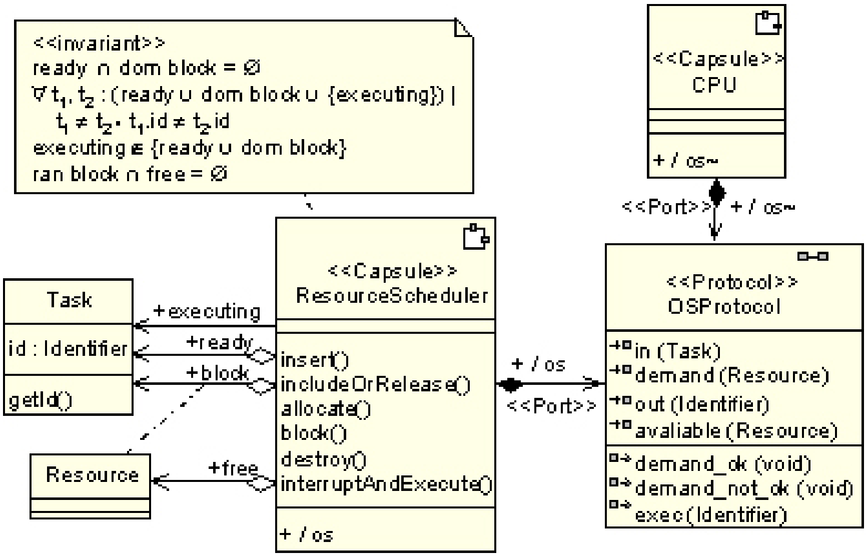


Fig. 1. ResourceScheduler Class Diagram

behavior) in communication of capsules. Ports can be public or private. Public ports allow communication with the environment, whereas private ports are used for communication with (and among) component capsules.

In general, a protocol may involve several participants (with several roles). Often, however, most applications are confined to binary protocols (involving only two capsules). For a binary protocol, only one role needs to be specified (the *base* role). The other complementary role, named *conjugate* and indicated with the suffix ~ in the base role, can be inferred by inverting inputs with outputs, and vice versa. A port which plays the conjugate role is represented by a white-filled (in opposition to a black-filled) square.

Connectors are used to interconnect two or more ports of capsules and thus describe their communication relationships. A connector is associated with a protocol and acts as a physical communication channel between ports which play complementary roles. A UML-RT class diagram typically includes capsules, classes and protocols of a system, and their relationships. As a convention, we describe invariants, pre- and pos-conditions as *notes* in the diagram. In Figure 1, we show a class diagram of a simplified operating system which is formed of two capsules: `CPU` and `ResourceScheduler`, that communicate through the `OSProtocol`. The port `os` of `CPU`, an instance of `OSProtocol`, has been set *conjugate* with respect to the `os` port of `ResourceScheduler`. It is worth noting that in the protocols input signals appear in the top compartment whereas output signals in the bottom compartment. For capsules the bottom compartment is used to declare ports; the remainder compartments

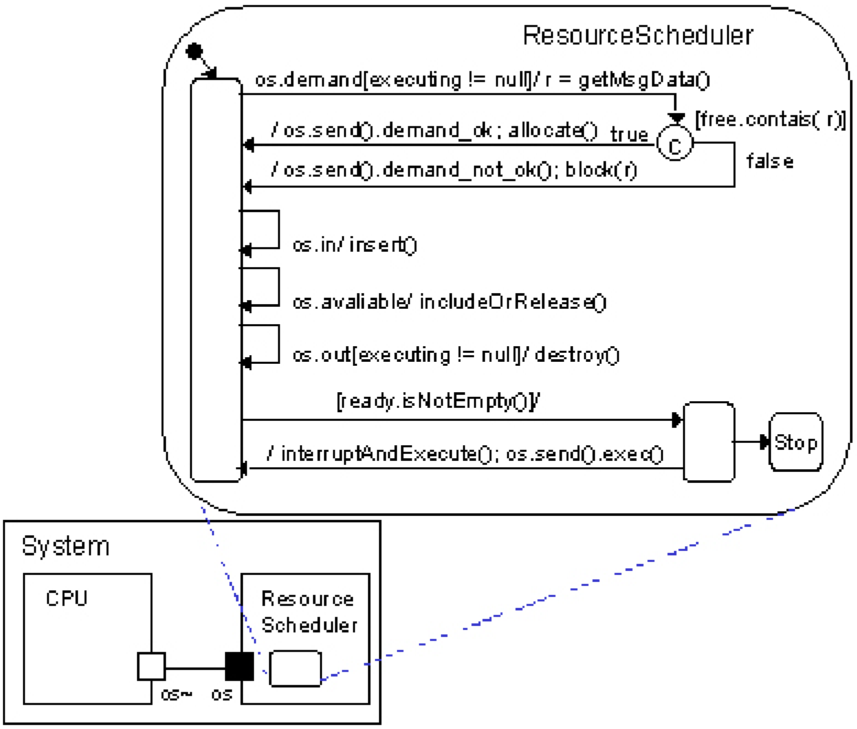

ResourceScheduler
os.demand[executing != null]/ r = getMsgData()
[free.contais(r)]
/ os.send().demand_ok ; allocate()
true
C
false
/ os.send().demand_not_ok(); block(r)
os.in/ insert()
os.avaliable/ includeOrRelease()
os.out[executing != null]/ destroy()
[ready.isNotEmpty()]/
/ interruptAndExecute(); os.send().exec()
Stop
System
CPU
Resource Scheduler
os~
os

Fig. 2. ResourceScheduler Structure Diagram

contain the same elements as in classes.

The attributes of `ResourceScheduler` are the task which is currently executing, a set of ready tasks, a set of tasks blocked on resources, and a set of free resources. The invariant states that: the sets of ready and blocked tasks are disjoint, tasks have distinct identifers, the executing task is not a ready nor a blocked task, and the set of free resources is disjoint from the resources on which tasks are blocked.

A view of the entire system can also be specified as a capsule that contains, as compound capsules, `ResourceScheduler` and `CPU`. A structure diagram represents this view, with connectors linking the ports of the capsules (Figure 2). As already mentioned, the actual behaviour of a capsule is given by a state machine diagram, as show in Figure 2 for the capsule `ResourceScheduler`, which reacts to events concerning scheduling and resource managing. In the figure, states are represented by rectangles and transitions by directed arrows. In general, a transition has the form `e[g]/a`, where `e` is an input signal (or a set of input signals), `g` is a guard and `a` is an action. The transition is triggered by the input signals and a true guard. As a result, the corresponding action is executed.

An action can be any composition of a signal output, a method call, or a command of the intended target programming language (for instance, Java [15]). As in the Rational Rose-RT [32], a signal output is expressed here using a `send` method, which may involve a parameter (the data transmitted with the signal). A signal input is represented just by the signal name (qualified by the corresponding port). If there is a parameter associated with the signal, the data is obtained via a method call (`getMsgData()`). For simplicity, we often omit signal parameters in the state machines.

The topmost transition of the state machine of Figure 2 triggers when the signal `demand` is received, and there is a task executing (`executing != null`). The action associated with this transition executes the method `getMsgData()` to obtain a parameter (in this case, a resource) sent with the signal `demand`. The subsequent transition depends on whether this resource is free or not. If it is free, the signal `demand_ok` is output and the new resource is stored in the set of free resources (through the action `allocate()`). Otherwise, the currently executing task is blocked on that resource.

Similarly, there are transitions to capture the insertion of a new (ready) task, release of tasks blocked on a resource, destruction and interruption of the executing task, and allocation of a ready task for execution.

In this paper we assume that communication via signals is synchronous. This is to simplify the mapping into the *OhCircus* language which, being based on CSP, allows only synchronous communication. Dealing with asynchronous communications is one of the topics for further investigation.

3 Laws for Classes and Capsules

The aim of the laws proposed here is to allow a systematic transformation of an analysis model into a more concrete UML-RT design model. Since a major concern is that these laws are useful for practitioners of software engineering, ideally both the laws and the associated side conditions should be purely syntactic. Furthermore, although we mention a few basic laws which capture very simple transformations, like introducing a new class, and attributes and methods to an existing class, our focus is on laws which express larger grain (design-level) transformations. We present a few laws which capture specific data refinement patterns, and characterise a more general data refinement technique annotating the models with formal constraints in the Z notation, as already mentioned.

In our refinement approach to UML-RT, inheritance (of both classes and capsules) must preserve behaviour [23]. Specific laws and a general technique for behavioural inheritance are presented. Investigating further refinement (and refactoring) patterns for data refinement and behavioural inheritance is one of the topics for future work.

In this work we interpret models more concretely than the view taken in Alloy [20] or UML/OCL. Here only directed associations are used, taken as class attributes, and not as relationships between all instances of classes. Thus association constraints are confined to class invariants. Furthermore, invariants can only include references to *visible* instances of the system environment, where the visible instances are those determined by the attributes of the class or by the attributes of another class accessed via navigation. An interesting and detailed investigation of the subtle problems which emerge during model transformations using UML with arbitrary constraints is reported in [12], which uses Alloy [20] as a formal basis.

While simple associations are interpreted as attributes, aggregation and composition are interpreted as data structures (in particular, as sets). When we write predicates we do not consider visibility issues, and reference (even private) attributes directly.

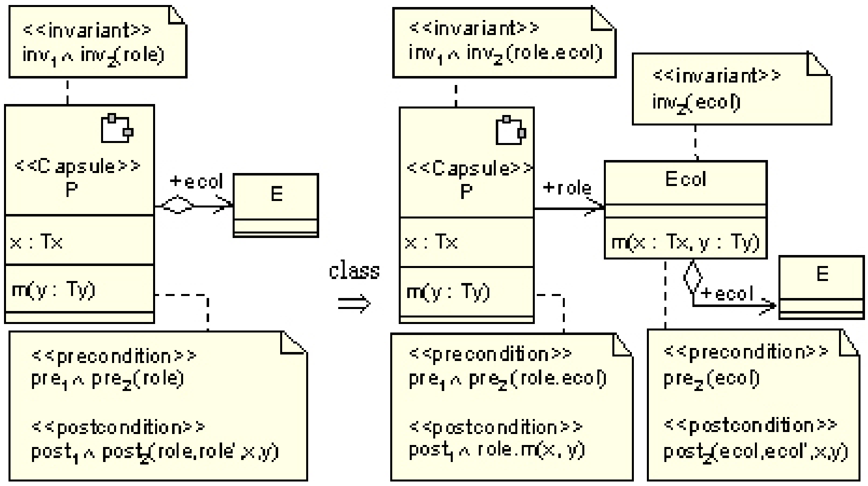
Apart from a relation on entire models (a graphical arrow) we use more specific relations on classes, capsules, and state machines to emphasise that some of the laws relate exclusively classes, and others express properties of capsules. These more specific relations are annotated in the obvious way. Some (if not all) the laws presented could be applied in both directions, but here we emphasise their use in refining an analysis into a design model. The side conditions presented concerns left to right applications.

3.1 Basic laws

We classify as basic laws those which express simple transformations such as introducing a new design element (attribute, method, class, capsule, protocol, relationship, ...) or expressing a general property of such elements. The granularity of such laws can be as small as just changing the visibility of some attribute or method, and are not the focus of this paper. Formalisation of these and many other basic laws for an object-oriented language can be found in [3], which also presents a normal form reduction strategy to illustrate the expressive power of the set of laws. Some basic laws for UML-like models expressed in Alloy is reported in [12].

3.2 Class extraction

It is common during analysis to identify a class (or capsule) which later, during design, is found to represent more than one abstraction. So a transformation to promote this hidden abstraction into an independent class is potentially



provided `Ecol` is a fresh identifier, and `role` is not free in $\mathtt{inv}_1$, $\mathtt{pre}_1$ and $\mathtt{post}_1$, and the state machine of `P` (assumed not to refer to `role`) is not modified

Fig. 3. Class extraction from a capsule

useful.

The transformation captured in Figure 3 seems recurrent during design steps. On the left-hand side of the figure, the capsule `P` has an aggregation relationship (represented by `role`) with a class `E`, standing for some entity type. A more concrete view is that `P` has an attribute `role` whose type is some data structure (here assumed to be a set) which stores elements of type `E`. Although the data structure provides some form of abstraction, the fact that the collection is, for example, implemented as a set is visible in `P`, while a more structured design (like a layered architecture) would isolate the collection as a separate abstraction, with its associated business rules. The right-hand side of the figure introduces the new class `Ecol` (representing an encapsulated collection of instances from `E`) and replaces the aggregation relationship between `P` and `E` with a simple association between `P` and `Ecol`, which itself has an aggregation relationship with `E`.

In order to ensure the validity of this transformation, we need to consider the invariant of `P` before and after the transformation, as well as and pre- and postconditions of methods in `P`. For simplicity, we single out the method `m`.

Concerning the invariant, consider that, before the transformation, it includes a predicate $\mathtt{inv}_2$ which may include references to `role`. As the type of `role` changes after the transformation, its occurrences are replaced with `role.ecol`. If the only free variable in $\mathtt{inv}_2$ is `role` itself, this predicate can be taken as the invariant of the class `Ecol`. There is flexibility on what should be the invariant of the class `Ecol`. Keeping the complete invariant in `P` (possibly with the modifications explained above) safe-guards the original contract and, therefore, from the perspective of the capsule `P`, the invariant of `Ecol` could be any predicate for which the methods of `Ecol` are feasible [27]. However, taking the relevant part of the invariant of `P` as the invariant of `Ecol` may be a more sensible decision, since, as a new abstraction, `Ecol` should have its appropriate contract. Similar considerations apply to pre-conditions of methods, as can be observed in the Figure 3. Nevertheless the pre-condition cannot be strengthened, as is well-known from standard date refinement [33].

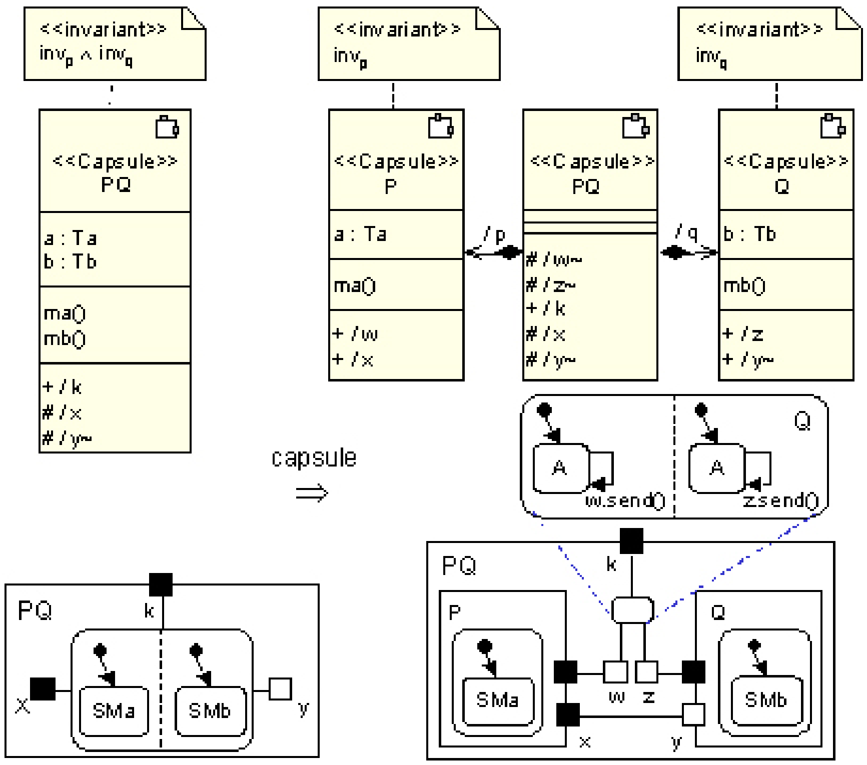
The effect of this law in each method of `P` is to possibly delegate part of the contract concerning the attribute `role` to a corresponding method in the new class `Ecol`. Assuming that the postcondition of `m` includes a predicate ($\mathtt{post}_2$) on initial and final values of `role`, this is delegated to the postcondition of the method `m` in class `Ecol`, replacing occurrences of `role` and `role'` with `ecol` and `ecol'`. In capsule `P`, the predicate $\mathtt{post}_2$ is replaced with a method call (`role.m(x,y)`). The additional parameter x is necessary since it is an attribute of `P` but not of `Ecol`. It may seem unusual method calls appearing in predicates. In our approach, however, this is not a problem because the language *OhCircus*, used as our model, has a well-defined semantics for predicates involving method calls.

All these conditions are necessary for extracting a class from another class. In the case of extracting a class from a capsule (as in Figure 3), we need to further consider the effect on the state machine of `P`. In general, the behaviour of the state machine must be preserved (or refined) by these transformations. The notion of refinement of state machines is discursed in the next section. For the purpose of this law, we require that state machine is not modified and that it does not refer to the attribute `role`.

3.3 Capsule decomposition

At an abstract analysis level, it is usually convenient to model an application using only a few components and interactions among them. During design, with increasing knowledge of the internal structure of components, it is often necessary to decompose these components into smaller ones, in order to tackle complexity and to potentially improve reuse.

When a component is decomposed, the subcomponents may be combined in a variety of ways, in order to preserve the original behaviour. For example, they can be combined in parallel, in sequence, non-deterministically, and so on; a process algebra like CSP offers a rich repertoire of operators to combine processes. Therefore we can think of a set of laws to decompose components: one for each operator, and possible combinations of operators.

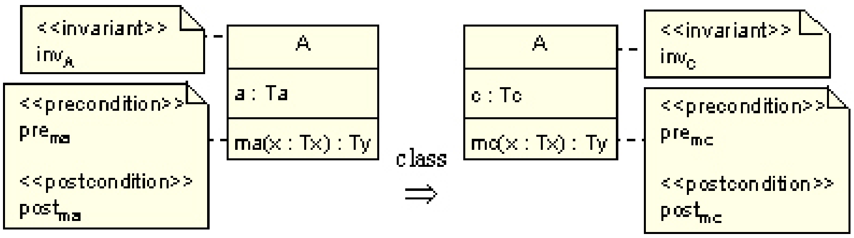


provided ⟨a, ma, x, SMa⟩ and ⟨b, mb, y, SMb⟩ partition PQ

Fig. 4. Parallel decomposition of capsules

In Figure 4, we introduce a law for decomposing a capsule into parallel component capsules. On the left-hand side of the figure, two diagrams represent different views (structure and behaviour) of the capsule to be decomposed, PQ. The state machine of PQ interacts with the external environment through the port k. This state machine is an *AND*-state composed of two states (SMa and SMb), which may interact (internal communication) through the conjugated ports x and y. Furthermore, in transitions on SMa, only the attribute a and the method ma is used, while transitions of SMb use only the attribute b and the method mb. Finally, the invariant of PQ is the conjunction $\texttt{inv}_p \wedge \texttt{inv}_q$, where $\texttt{inv}_p$ involves only a as free variable, and $\texttt{inv}_q$ only b. When a capsule obeys such conditions, we say that it is partitioned. In this case, there are two partitions: one is ⟨a, $\texttt{inv}_p$, ma, x, SMa⟩ and the other is ⟨b, $\texttt{inv}_q$, mb, y, SMb⟩.

The effect of the decomposition is to create two new component capsules, P and Q, one for each partition, and redesign the original capsule PQ to act as a controller. The new behaviour of PQ depends on the particular form of decomposition. Figure 4 captures a parallel decomposition. The structural



provided 1. $\forall$ a : Ta; c : Tc; x : Tx | $\texttt{inv}_A \wedge \texttt{inv}_C \bullet \texttt{pre}_{ma} \wedge \texttt{CI} \Rightarrow \texttt{pre}_{mc}$

2. $\forall$ a : Ta; c, c' : Tc; x : Tx; y : Ty | $\texttt{inv}_A \wedge \texttt{inv}_C \wedge \texttt{inv}_C\texttt{'} \bullet$
$\texttt{pre}_{ma} \wedge \texttt{CI} \wedge \texttt{post}_{mc} \Rightarrow (\exists\, \texttt{a' : Ta} \mid \texttt{inv}_A\texttt{'} \bullet \texttt{CI'} \wedge \texttt{post}_{ma})$

Fig. 5. Data refinement for classes

diagram on the right-hand side of the figure shows the three state machines, and how their ports are connected. The new state machine of PQ has the same structure as the original one, except that now its role is only to serve as a (parallel) *proxy* between the external port k and the internal capsules P and Q. The new ports w and z were created to connect the state machine of PQ with those of P and Q, respectively. The union of the signals of protocols W and Z is the set of signals of protocol K. However, there is no need for W and Z to be a partition of K: intersection of signals in W and Z is allowed.

3.4 Data Refinement

Both laws presented in the previous sections embody some change of data representation. However, as they deal with specific data transformations, captured by the syntactic patterns involved, the well-known proof obligations associated with data refinement (like applicability and correctness) are implicitly discharged.

In the general case, however, it may be necessary to carry out changes of data representation in contexts not covered by a specific pattern. So it is important to characterise the general notion for UML-RT classes and capsules.

For classes, we adapt the familiar data refinement technique of Z [35], with proof obligations to ensure applicability and correctness (see the proviso of Figure 5, where CI stands for the coupling invariant). Although the law suggests that the original class attributes and methods are renamed, this is just to avoid confusion when expressing the proof obligations. Figure 5 illustrates this by preserving the name of the class, which is not referenced in the proviso, and renaming its attributes and methods.

The data refinement technique for capsules is based on that for classes, but further considering the impact on the capsule state machine, which is an object-oriented extension of a statechart of Harel [16,17]. The semantics of such machines can be given by hierarchical labelled transition systems (HLTS), which are modular LTSs[4]. In general, however, due to the complexity involved on analysing such hierarchies, one firstly transforms an HLTS into a *flat* LTS (FLTS) [24], or simply LTS for short. In the remainder of this paper we consider that a capsule state machine is an LTS, whenever convenient.

Following the works of Harel [17] and Milner [26], we postulate that refinement for capsules ($\mathtt{P} \overset{capsule}{\Rightarrow} \mathtt{Q}$) always holds when the state machine of Q simulates the behaviour of P. In the notation of [17], this simulation relation is written as ($\mathtt{SM}_Q \preceq \mathtt{SM}_P$). Here we write it as $\mathtt{SM}_Q \overset{machine}{\Rightarrow} \mathtt{SM}_P$.

A simpler proof technique for data refinement of capsules assumes class refinement between the two capsules, and the following notion of compatibility between their state machines.

Definition 3.1 (**Compatibility**) Let P and Q be two UML-RT capsules, such that $\mathtt{P} \overset{class}{\Rightarrow} \mathtt{Q}$, for some coupling invariant CI. We say that the state machine of P ($\mathtt{SM}_P$) is compatible with the state machine of Q ($\mathtt{SM}_Q$), `compatible(P,Q)`, iff $\mathtt{SM}_P$ has the same LTS of $\mathtt{SM}_Q$, except that for each transition $e_P\,[g_P]\,/\,a_P$ of $\mathtt{SM}_P$, $\mathtt{SM}_Q$ has the corresponding transition $e_Q\,[g_Q]\,/\,a_Q$ such that

- If r is a simple request (event) then ($r \in e_P \Leftrightarrow r \in e_Q$) and ($r \in a_P \Leftrightarrow r \in a_Q$);
- If `ma()` is a method call in $\mathtt{SM}_P$ then ($\mathtt{ma()} \in e_P \Leftrightarrow \mathtt{mc()} \in e_Q$) and ($\mathtt{ma()} \in a_P \Leftrightarrow \mathtt{mc()} \in a_Q$), for some method `ma` of P and its refinement `mc` of Q;
- $\forall\, a : Ta;\ c : Tc;\ x : Ty \mid CI \bullet (g_P \Leftrightarrow g_Q)$;

□

Therefore, if $\mathtt{P} \overset{class}{\Rightarrow} \mathtt{Q}$ and `compatible(P,Q)` then $\mathtt{P} \overset{capsule}{\Rightarrow} \mathtt{Q}$.

3.5 Behavioural inheritance

Generalisation is a very important design tool. It is common during the development process to factor out an abstraction which can then be reused to (re)define several others, more specialised ones, by inheritance. Conceptually, this should not be confined to code reuse. The behaviour of the superclass must be preserved by the subclass, whenever formal refinement is a major concern, as is our purpose here. This is precisely captured by the notion of behavioural inheritance [23,17]. For classes, it can be characterised in terms of standard data refinement. If B is introduced as a subclass of A, this generates a proof obligation that B must data refine A, concerning the original methods of A.

[4] An LTS is basically a directed graph capturing the behaviour of a concurrent system.

In [23], the *principle of substitutability* is well-motivated and formally characterised to precisely capture under what conditions an object of a class can be replaced with an object of the subclass, preserving the original behaviour. This characterisation is in terms of three main notions: behaviour of methods, invariants and history properties. Another important issue is signature compatibility (based on *contravariance* of arguments and *covariance* of results of methods [23]).

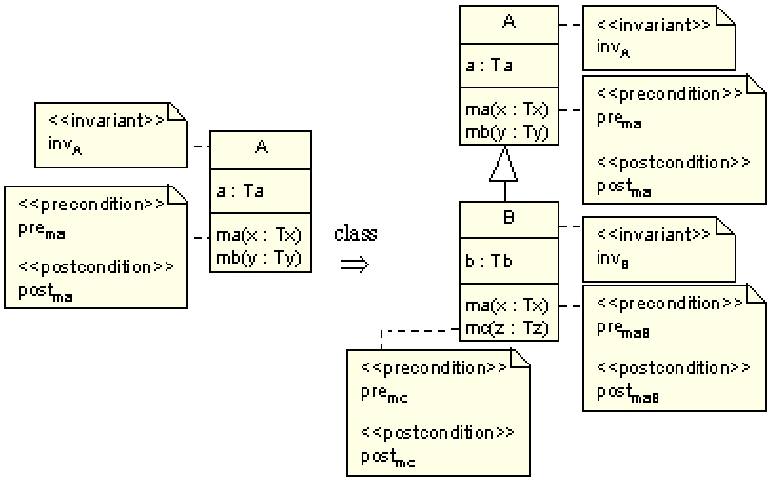
Ensuring behavioural inheritance of redefined methods follows the same conditions for data refinement. Concerning the invariant, the one of the subclass must be at least as strong as the one of the superclass. Furthermore, new methods must preserve the invariant. Regarding history properties, they impose constraints over state pairs during the computation lifecycle of an object. Preservation of such properties by a subclass requires that they not weakened.

In our approach we adopt the characterisation of [23], as captured by Figure 6, but we do not consider history properties. The reason is that such properties can be violated only in the presence of sharing, which we will consider as future work. In the law presented in Figure 6 we also left implicit conditions of signature compatibility, as these are easily verified at the level of typechecking.

In many cases during practical modelling, the proof obligations of the law presented in Figure 6 can be simplified. For instance, consider that the coupling invariant between the super and the subclass is the identity function (`A.a` =B.a+). This states that, for two related objects of type `A` and `B`, their attribute `a` has exactly the same value. In this case, the proof obligations could be simplified by removing the occurrences of `CI`. Another source of simplification is when `inv_{B}` = `inv_{A}`, and the pre-condition of methods is maintained. In this case, only the last obligation would need to be proved.

Capsule inheritance was initially defined in ROOM [1], indicating that besides structural inheritance (attributes, methods and ports) the supertype behaviour (defined by the capsule state machine) is also inherited. In the definition of ROOM, the inherited state machine can be freely modified. As we are concerned with behavioural preservation, we need to associate some proof obligations with capsule inheritance, as we have done for classes, that can similarly be characterised in terms of data refinement. We define capsule inheritance as an extension of behavioural class inheritance by adding new constraints on the state machines.

For Capsule we will assume the extension of the theory of substitutability, in the work of Harel [17], for actives classes (capsules) on state based systems. There, if all behaviour (contained in the state machine) of a parent capsule is somehow present in the inheriting capsule, then the parent capsule is simulated



provided $1.\mathtt{inv}_B \wedge \mathtt{CI} \Rightarrow \mathtt{inv}_A$

$2.\mathtt{pre}_{ma} \wedge \mathtt{CI} \Rightarrow \mathtt{pre}_{maB}$

$3.\mathtt{post}_{maB} \wedge \mathtt{CI} \Rightarrow \mathtt{post}_{ma}$

Fig. 6. General pattern of behavioural subclassing

by this inheritance. According to Harel, the notion of simulation leads to the notion of substitutability.

In our work, we regard capsule inheritance of state machines as an AND-state formed of the state machines of the capsule and the one of the subcapsule, as in the approach defined for CSP-OZ [10]. Additionally, we observe that a capsule can offer more services than its supertype, and possibly additional ports. We require that the inherited ports preserve their original signatures (exactly the same types of parameteres). An interesting topic for further investigation is to relax this condition and allow contra- and covariance of port parameters, as discussed above for methods.

4 Case study: a resource scheduler

In order to illustrate the application of the laws proposed in the previous section, we refine the abstract model of the `ResourceSheduler`, presented in Section 2 (see Figure 1), into a more concrete design model. Each subsection below is dedicated to a relevant refinement step.

4.1 Resource and Scheduler partitions

Aiming at decomposing the capsule `ResourceScheduler` into the component capsules `ResourceManager` and `Scheduler`, the first step is to transform the

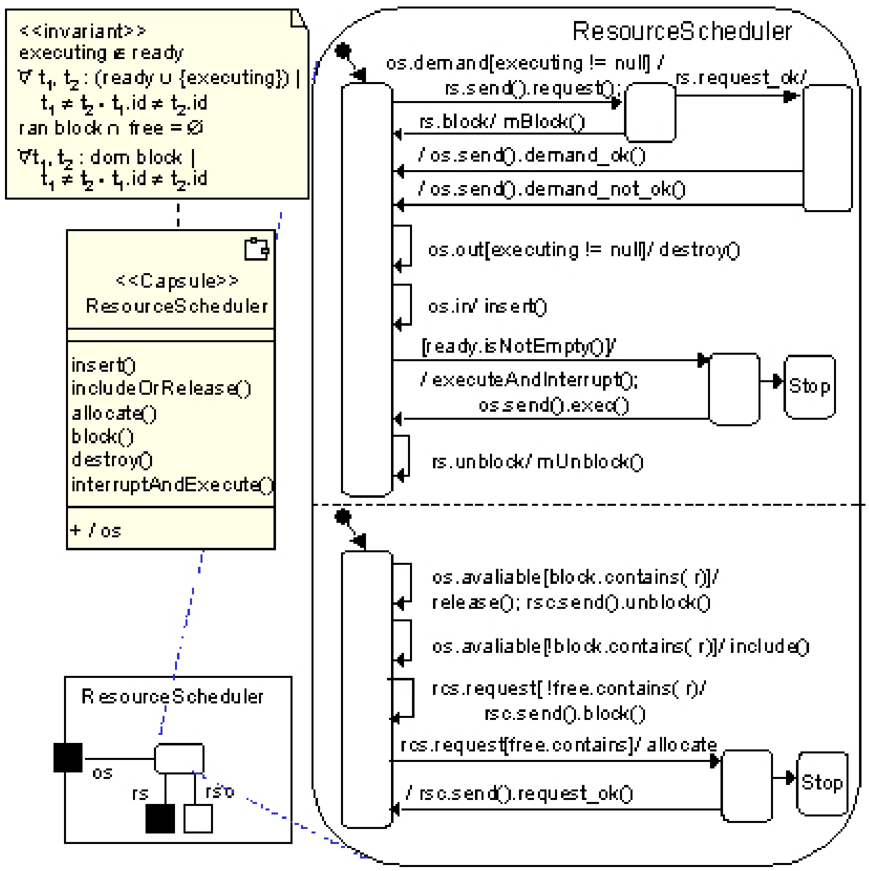

<<invariant>>
executing ∈ ready
∀ t1, t2 : (ready ∪ {executing}) | t1 ≠ t2 • t1.id ≠ t2.id
ran block ∩ free = ∅
∀t1, t2 : dom block | t1 ≠ t2 • t1.id ≠ t2.id
<<Capsule>>
ResourceScheduler
insert()
includeOrRelease()
allocate()
block()
destroy()
interruptAndExecute()
+ / os
ResourceScheduler
os
rs
rsc
ResourceScheduler
os.demand[executing != null] / rs.send().request();
rs.request_ok/
rs.block/ mBlock()
/ os.send().demand_ok()
/ os.send().demand_not_ok()
os.out[executing != null]/ destroy()
os.in/ insert()
[ready.isNotEmpty()]/
/ executeAndInterrupt(); os.send().exec()
Stop
rs.unblock/ mUnblock()
os.avaliable[block.contains(r)]/ release(); rsc.send().unblock()
os.avaliable[!block.contains(r)]/ include()
rcs.request[!free.contains(r)/ rsc.send().block()
rcs.request[free.contains]/ allocate
/ rsc.send().request_ok()
Stop

Fig. 7. ResourceScheduler State Machine

state machine of ResourceScheduler (see Figure 2) into an AND-state machine which reflects the capsule partitioning. As shown in Figure 7, the top AND-state is concerned with the signals related to scheduling activities; this state machine refers only to attributes and methods which are relevant for scheduling. The bottom AND-state captures the resource manager activities, and refes only to the corresponding signals, attributes and methods.

With this partitioning, the two AND-states need to communicate in order to preserve the effect of the original state machine. For instance, the unblock operation is now specified as a cooperation of both AND-states. When the bottom state machine receives, from the environment, the signal `available` (and a resource), it recovers the tasks blocked on that resource and sends these tasks (as a parameter of signal `unblock`) to the top state machine, which adds them to the set of `ready` tasks (executing method `mUnblock`).

The refinement above is a pure (although not at all trivial) state machine

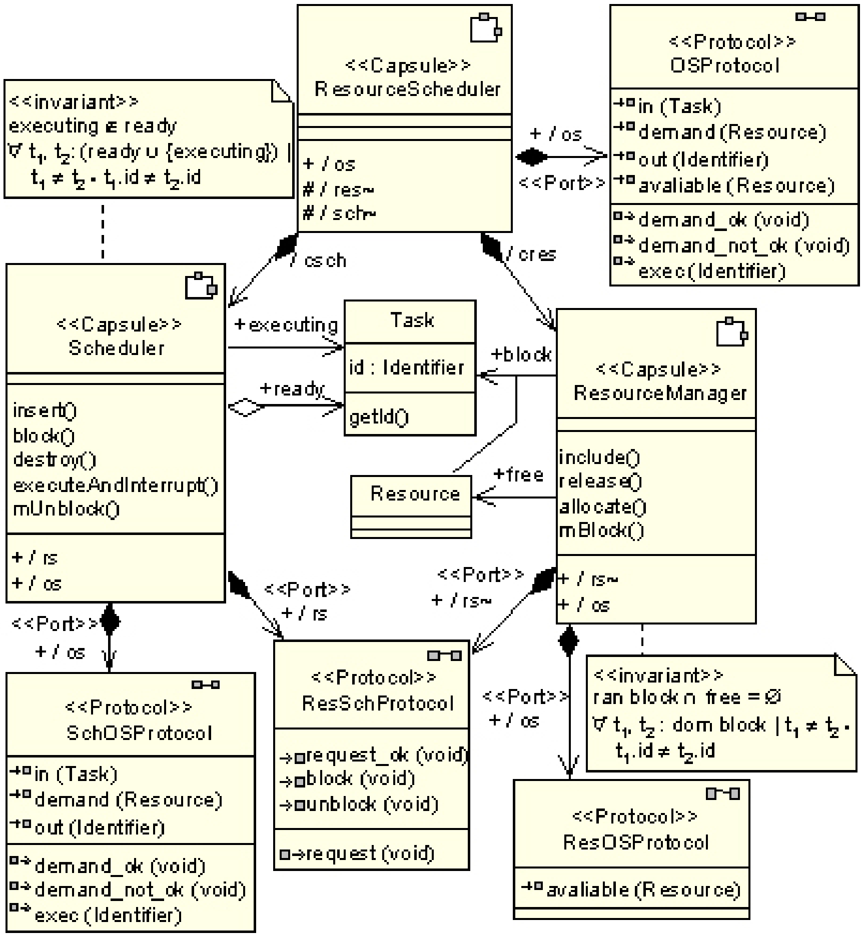


Fig. 8. Class diagram of ResourceScheduler decomposition

refinement. Formally, it must be carried out using the simulation relation [17] of state machines, as discussed in the previous section. This detailed refinement is out of the scope of this paper.

Apart from the state machine refinement, this step embodies a simple, but subtle, data refinement to weaken the `ResourceScheduler` invariant, so that each predicate in the invariant refers exclusively to attributes related to scheduling or to resource managing, but not both. Recall that this is one of the necessary conditions to allow the decomposition of the capsule, which is the subject of the next section.

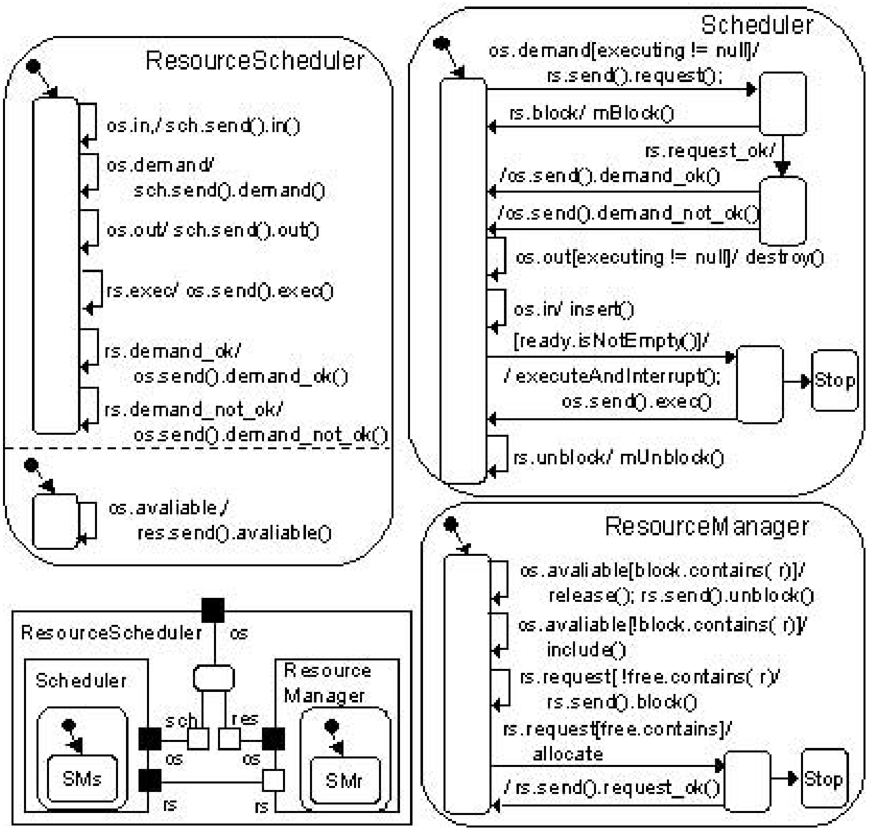


Fig. 9. State machines of ResourceScheduler, Scheduler and ResourceManager

4.2 Capsule decomposition: Resource and Scheduler capsules

As we can now identify two clear partitions in the capsule `ResourceSheduler`, it is possible to apply the parallel decomposition law (Figure 4) and generate the `ResourceManager` and `Scheduler` capsules (Figure 8).

The capsule `ResourceSheduler` is associated with the `OSProtocol`, as before. The two sets of communication signals for interaction of `Scheduler` and `ResourceManager` with the environment partition the `OSProtocol`. The latter (singleton) set gives rise to the `ResOSProtocol`, and the former is captured by the `SchOSProtocol`. Apart from these protocols, the `ResSchProtocol` embodies the signals for the communication between `ResourceManager` and `Scheduler`.

The new state machine of `ResourceSheduler` acts as a proxy which activates the two component capsules in parallel, as presented in Figure 9.

4.3 Data refining Scheduler: extracting a task collection

The remaining steps of our case study focus on a more concrete design for the capsule `Scheduler`. In this step we refine the `ready` association role from an aggregation of tasks into a simple association with a new `TaskCollection` class, as shown in Figure 10.

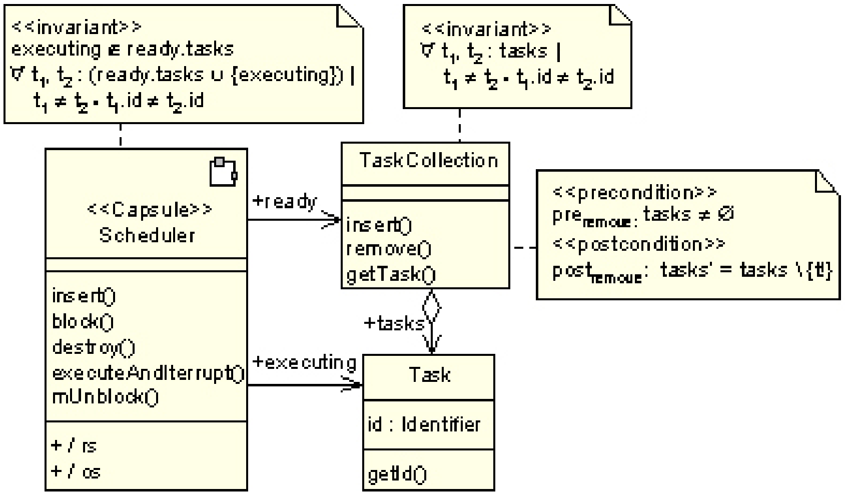


Fig. 10. Extracting TaskCollection class from Scheduler

This can be justified by a direct application of the class extraction law, presented in Figure 3. To ensure that the capsule `Scheduler` really matches the left-hand side of the law, we would need to explicit the class invariant and the pre- and postcondition of each method, which we omit for limitation of space.

4.4 Introducing priority tasks

The target scheduler of our development is one which allocates tasks based on priority. The purpose of this step is to introduce the class `PriorityTask` which inherits from `Task` and includes a new attribute to record priority. We also introduce `PriorityTaskCollection` as a subclass of `TaskCollection`.

In principle, this should not be necessary, since `TaskCollection` itself might store instances of `PriorityTask`, but the inherited collection is introduced because it redefines the `remove` method responsible for selecting the next element of the collection for scheduling: while in the original collection this choice is totally arbitrary, in the inherited one the task with higher priority is returned. Figure 11 shows the new classes and their relationships.

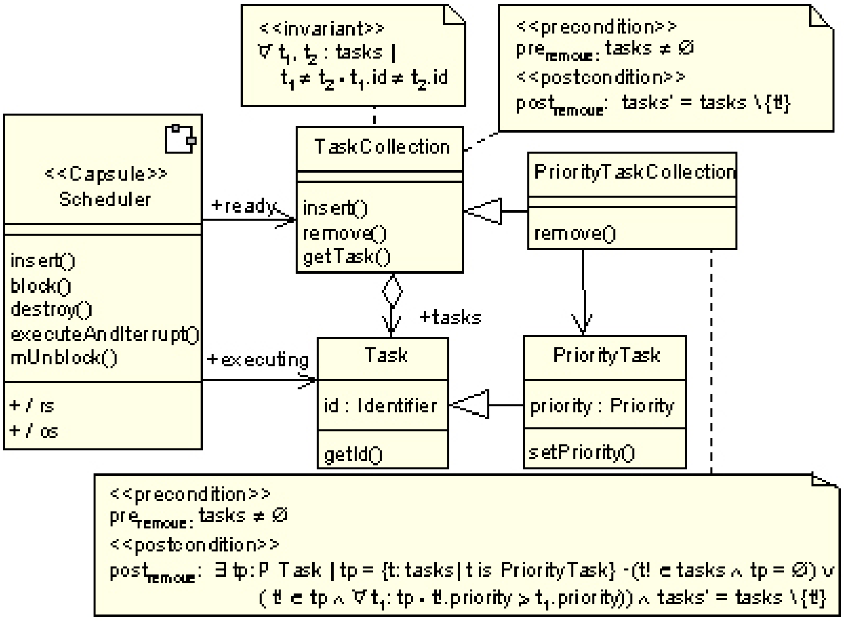


Fig. 11. Introducing PriorityTask and PriorityTaskCollection

As already mentioned, this generates proof obligations to ensure behavioural inheritance, which entails proving that the subclass data refines its superclass for some coupling invariant relation. In this particular case, in both refinements, the pattern captured by the law presented in Figure 6 is applicable. In the case of `PriorityTask`, the proof obligations as easily discharged, since it involves no method redefinition and the extra method `setPriority` changes only the new attribute. Concerning `PriorityTaskCollection`, only `remove` is redefined. Therefore, in both cases, the only condition is check is the strengthening of the associates postconditions. This can be discharged by simple predicate calculation.

4.5 A priority scheduler

A capsule `PriorityScheduler` can be designed as a specialisation of the parent capsule `Scheduler`. The effect is inclusion of attributes and methods of `Scheduler` into `PriorityScheduler` (with conjunction of invariants, and of postconditions of redefined methods). In our example, no method needs to be explicitly redefined, only the invariant is strengthened, as presented in Figure 12. Concerning the state machine, recall that the semantics of capsule inheritance is an AND-state formed of the state machines of the capsule and the one of the subcapsule. As `PriorityScheduler` has no additional control

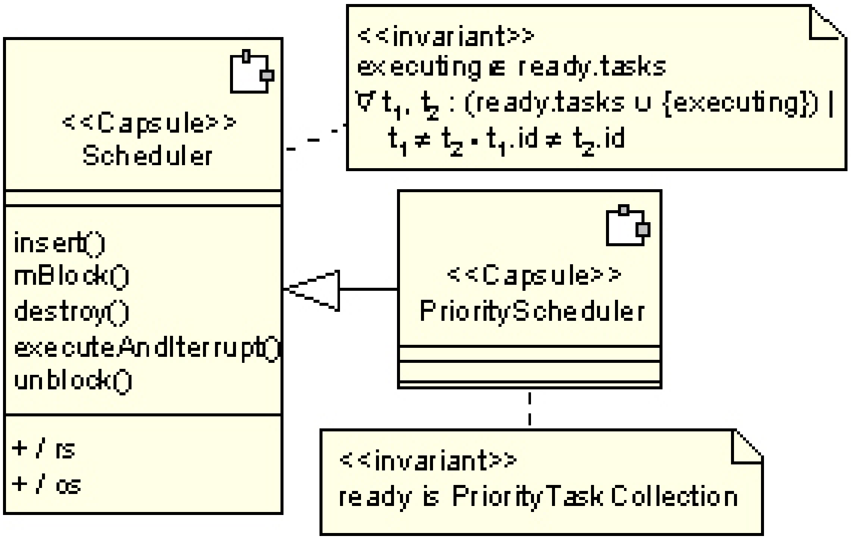


Fig. 12. PriorityScheduler as a subcapsule of Scheduler

flow behavior, its state machine is just that of `Scheduler`.

5 Translation into *OhCircus*

In this section we suggest that a possible alternative to address the soundness of the laws proposed for UML-RT is through a mapping into *OhCircus*. The motivation for this mapping is that the main design elements and refinement notions presented for UML-RT have a formal counterpart in *OhCircus*, based on the unifying theories of programming.

We show how an *OhCircus* specification emerges gradually from our case-study. We provide a rather informal translation from UML-RT into *OhCircus* because this is also a theme for further research.

5.1 UML-RT classes

We start the translation with a very simple UML-RT class: `Task` (see Figure 8). This generates a class in *OhCircus* with the same attributes and methods, as expected.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Task} \mathrel{\widehat=} \textbf{begin} \\[1em]
\quad \textbf{state}\ \mathit{TState} \mathrel{\widehat=} [\, \mathit{id} : \mathit{Identifier} \,] \\[1em]
\quad \textbf{public}\ \mathit{getId} \mathrel{\widehat=} \textbf{res}\ n : \mathit{Identifier} \bullet n := \mathit{id} \\[1em]
\textbf{end}
\end{array}
$$

Although it is not visible from the diagram in Figure 8, the body of the method—a Z predicate—comes directly from the UML-RT `Task` class annotations.

5.2 UML-RT capsules

Differently from classes, UML-RT capsules correspond to *OhCircus* processes. An *OhCircus* process is like a class except that a behaviour can also be described via *actions* which are combined using the CSP operators.

The capsule `Scheduler` (see Figure 8) is translated into the following process.

$$
\begin{array}{l}
\textbf{process}\ \mathit{Scheduler} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{ASState} \mathrel{\widehat=} \\[1em]
\qquad\quad [\, \textbf{private}\ \mathit{executing} : \mathit{Task};\ \textbf{private}\ \mathit{ready} : \mathbb{P}\, \mathit{Task} \mid \ldots \,] \\[1em]
\quad \textbf{initial}\ \mathit{ASInit} \mathrel{\widehat=} [\, \mathit{ASState}' \mid \mathit{executing}' = \textbf{null} \land \mathit{ready}' = \emptyset \,] \\[1em]
\quad \textbf{private}\ \mathit{Insert} \mathrel{\widehat=} [\, \ldots \,]
\end{array}
$$

Its state space is formed of two attributes whose names are the same as those of the associations in the model. Furthermore, they are private, as is usual in the design of a capsule. The invariant and method bodies are omitted, as they are directly obtained from the annotations in the UML-RT model; only the body of the initialisation method is presented for illustration. Like attributes, methods are also private.

After describing the static part, we consider the dynamic one which is expressed by *OhCircus* actions. We capture the main state machine by the action named *Machine* (defined in the next section). The complete behaviour is given by the following main action.

- $(\mathit{ASInit};\ \mathit{Machine}) \setminus \mathit{protPorts} \cup \mathit{connectors}$

end

where *protPorts* and *connectors* are sets of channels representing the protected ports and connectors used inside the capsule, respectively. Since protected ports and inside connectors have the scope of the capsule and *OhCircus* channels have a global scope, we need to hide ($\setminus$) them ($\mathit{protPorts} \cup \mathit{connectors}$) from outside the capsule.

5.3 UML-RT state machines

We now present a definition for the action *Machine* based on the state machine depicted in Figure 9. From the start state we can engage in the transition

```
os.demand [executing ≠ null] / r = getMsgData();
          rs.send().request(executing, r)
```

as long as the signal `os.demand` is received and the guard `executing` $\neq$ *null* is true. After that, the action `rs.send().request()` should be executed and the machine goes to a next state. From this transition we know that `demand` is an input signal, where the input value is catched by `r = getMsgData()` (see Section 2), and `request` is an output signal, sending the message `(executing, r)`. Thus by considering *Machine* and S_1 as the start and the next states, respectively, the previous behaviour can be partially expressed in CSP as

$$\mathit{Machine} \mathrel{\widehat=} \mathit{executing} \neq \mathit{null} \ \&\ \mathit{demand}?r \rightarrow \mathit{request}!\mathit{executing}!r \rightarrow S_1$$

At this point we have two choices: `request_ok` or `block`. From the previous mapping and due to the fact that state machines do not have explicit internal choices we get

$$S_1 \mathrel{\widehat=} \mathit{request\_ok} \rightarrow S_2 \,\Box\, \mathit{block} \rightarrow \mathit{Block}();\ \mathit{Machine}$$

The new state S_2 can be straightforwardly defined as

$$S_2 \mathrel{\widehat=} \mathit{demandOk} \rightarrow \mathit{Machine} \,\Box\, \mathit{demandNotOk} \rightarrow \mathit{Machine}$$

and so on. In statechart terminology such transformations apply only to OR-states, where there is only one active state at a time.

However, state machines can be more complex as illustrated in Figure 7. An AND-state formed of substates SM_{up} and SM_{down} is translated into the following *OhCircus* action

$$ResourceScheduler \mathrel{\widehat=} SM_{up} \; [\![X \mid\mid Y]\!] \; SM_{down}$$

such that $ResourceScheduler$ must behave as an interleaving of SM_{up} and SM_{down} when the triggers available are needed by SM_{up} and SM_{down}, simultaneously, and synchronously when some trigger of SM_{up} is waiting for an action of SM_{down} and vice-versa.

To obtain such a behaviour we first assume that $evt(P)$ results in all triggers of all transitions of P and thus define X to be

$$evt(SM_{up}) \setminus (evt(SM_{up}) \cap evt(SM_{down}))$$

that is, all triggers of SM_{up} except those common to SM_{up} and SM_{down}, and Y to be

$$evt(SM_{down}) \setminus (evt(SM_{up}) \cap evt(SM_{down}))$$

that is, all triggers of SM_{down} except those common to SM_{up} and SM_{down}.

5.4 Ports, protocols, and connectors

We make no distinction between ports and protocols in *OhCircus*. From the protocols, we map the signals and respective types into channel declarations, as follows.

$$\textbf{channel } demand, available : Resource$$

$$\textbf{channel } block, unblock, request, request\_ok$$

Finally, connectors are simply viewed as channels as well. Thus following [11], suppose that a connector `c` be connecting two ports `p` (of capsule `P`) and `q` (of capsule `Q`). Then, in *OhCircus* we have the following representation

$$P[\![c/p]\!] \; [\![\mid c \mid]\!] \; Q[\![c/q]\!]$$

where the channels c, p, and q must be compatible (same type).

5.5 Linking refinement relations

By translating a UML-RT model into *OhCircus*, as briefly illustrated above, one can then benefit from the formal notions and refinement laws of *OhCircus*. In particular, for each refinement relation used in our strategy for refining classes and capsules in UML-RT, there is a corresponding *OhCircus* relation which assigns meaning to it. For example, the meaning of capsule refinement (P $\overset{capsule}{\Rightarrow}$ Q) is process refinement in *OhCircus*(P $\sqsubseteq_{\mathcal{P}}$ Q). Similarly, class refinement in UML-RT is defined as class refinement in *OhCircus*, state machine refinement is defined as action refinement, and refinement of an entire model is program refinement in *OhCircus*.

Concerning data refinement, for UML-RT we have focused on a proof technique. In a formal language such as *OhCircus*, the technique is usually proved sound from a more general definition of simulation. So the standard data refinement technique which we adapted for UML-RT classes coincides with one of the laws of the *OhCircus* simulation relation: the law for schema actions.

Once these connections among the refinement relations are established, it is possible, for example, to relate laws expressed in the two notations, and carry out their proofs in *OhCircus*. As an illustration, the capsule decomposition law presented in Figure 4 has been directly inspired by the following law [30] originally presented for *Circus*, which therefore is also a law of *OhCircus*.

In what follows, we assume that pd stands for the process declaration below, where we use $Q.pps$ and $R.pps$ to stand for the process paragraphs of the processes Q and R; and F for a context (function on actions) which must also make sense as a function on processes (according to the *Circus* syntax).

$$\begin{array}{l}
\textbf{process } P \mathrel{\widehat=} \textbf{begin} \\
\quad State \mathrel{\widehat=} Q.st \land R.st \\
\quad Q.pps \upharpoonright R.st \\
\quad R.pps \upharpoonright Q.st \\
\quad \bullet\ F(Q.act, R.act) \\
\textbf{end}
\end{array}$$

The state of P is defined as the conjunction of two other state schemas: $Q.st$ and $R.st$. The actions of P are $Q.pps \upharpoonright R.st$ and $R.pps \upharpoonright Q.st$, which handle the partitions of the state separately. In $Q.pps \upharpoonright R.st$, each schema expression in $Q.pps$ is conjoined with $\Xi R.st$. This means that these process paragraphs do not change the state components of $R.st$; similarly for $R.pps \upharpoonright Q.st$.

The law below applies to processes in the above form.

Law 5.1 (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, and pd stand for the process declaration above. Then

$$pd = (qd\ rd\ \mathbf{process}\ P \mathrel{\widehat=} F(Q, R))$$

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

The above law can be informally justified by first adding the declarations qd and rd to the left-hand side, and then by promoting the context F from main actions to the corresponding processes. Formally, it is a simple consequence of the semantics of process combinators in *Circus* [6].

It is worthwhile contrasting the presentation of the law presented in Figure 4 and the above law. The behaviour of a process in *Circus* and in *OhCircus* is defined by an action, which can be expressed using CSP operators. Since most CSP operators are also available for combining processes, a process decomposition law can be uniformly and elegantly captured by a form of lifting the relevant operator from actions to processes, as presented in [5]. In UML-RT, even considering that we can use some statecharts combinators like AND-state, no operator exists to compose capsules; thus the composition of interest (parallelism, choice, ...) is achieved using a controller state machine (a proxy), and protocols which ensure the expected synchronisation between the controller and the component capsules (see decomposition law for UML-RT in Figure 4). This makes the presentation of the laws relatively more dense than in a notation in which the combination of the decomposed elements is totaly embedded into an appropriate operator.

6 Conclusion

Based on results obtained with formal notations and techniques, we proposed some transformation laws for UML-RT which seem useful to support a refinement strategy from analysis to design models, involving classes, capsules, protocols and constraints expressing invariants, pre- and postconditions.

The main inspiration for this work has been the research on *Circus* and *OhCircus*, and particularly the refinement strategy developed for process refinement in *Circus* [30,5], which is being currently extended for considering class refinement and behavioural inheritance as available in *OhCircus*. This has naturally led us to proposing the use of *OhCircus* as the semantic model for our work with UML-RT.

The attempt to migrate and possibly extend results from formal methods to semi-formal notations and processes (like UML-RT and the Rational Unified Process) seems interesting in itself. But making these results useful in practice is the real challenge. In this direction, we believe that providing a library of transformation laws for the practitioner developer is a significant contribution. Of course this must be backed upon a sound mathematical basis.

In our particular experience with UML-RT and *OhCircus*, some features of these two languages can be compared and contrasted. While both languages provide notions of classes, processes (capsules) and communication (protocols and channels), *OhCircus* have been carefully designed to support a uniform refinement stretegy, which does not seem the case with UML-RT. This has been briefly illustrated in the previous section, where we can note the difference in expression of the decomposition laws for capsules (processes). This difficulty is perhaps also reflected into processes which support the notation. For instance, the Rational Unified Process [21] (RUP) embodies some activities for identifying capsules and designing their internal structure. One of the steps related to capsule identification requires that the developer decide whether the identified capsule has a single or multiple flows of control, giving rise to a single or multiple capsules, respectively. Our view is that this step is too early in the process, when the developer does not yet have a precise view of the control structure of a capsule. Therefore, in RUP there is no support for a systematic decomposition of capsules, as we propose here.

Regarding UML software development by transformations, the work reported in [22] gives a formal semantics to UML, OCL, and statecharts, in terms of the Real-time Action Logic (RAL). From this semantics, a number of laws for UML development is proposed and proved. The work reported in [13] also addresses model transformations, but instead of using a logic it uses graph representations. With respect to refactoring, [8,9] propose a formal semantics to the structural part of UML and use that semantics to prove derived properties from UML class diagrams. The properties derivation is accompanied by changes on the class diagram. Nevertheless, none of these works address transformations involving capsules.

One immediate topic for future work is to address the soundness of the proposed laws in detail, with a complete mapping of all the notation of UML-RT (as well as the refinement relations for state machines, classes, capsules and entire models) to *OhCircus*. This requires further work on the *OhCircus* side as well, whose semantics is being currently defined using the unified theories of programming. An alternative to the mapping of UML-RT into *OhCircus* could be the definition of a semantics based on the unified theories of programming to UML-RT itself. Considering asynchronous communication is also a relevant topic for investigation, since this seems more usual among developers who use UML-RT.

Although we have proposed general techniques for data refinement and behavioural inheritance of classes and capsules, the real interest is on a comprehensive set of laws for model refinement and refactoring. In particular, while code refactoring for classes is extensively discussed in the literature, model refactoring for classes is not, let alone for capsules.

Process and tool support for the proposed refinement approach is clearly important. An adaptation of RUP with specific activities to guide a systematic migration from an analysis into a more concrete design model, possibly involving several levels of capsule decomposition, seems an interesting topic of study to pursue. A *plug-in* or *add-on* to the Rational Rose-RT to support this process would also help to motivate practical use.

Extending the approach with sharing (references) and time (both on the UML-RT side as well as on the *OhCircus* side) is a longer term plan.

Acknowledgement

We owe much to Ana Cavalcanti and Jim Woodcock for the joint work on *Circus* and *OhCircus*, which has been the main source of inspiration for this work. The case study presented here in UML-RT has previously been specified in *OhCircus* by Ana Cavalcanti, Augusto Sampaio and Jim Woodcock. We also thank Paulo Borba, Márcio Cornélio, Rohit Gheyi, Tiago Massoni and Franklin Ramalho for discussions related to this work.

References

[1] David Agnew, Luc J. M. Claesen, and Raul Camposano, editors. *An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems*, volume A-32 of *IFIP Transactions*. North-Holland, 1993.

[2] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[3] Luca Cardelli, editor. *A Refinement Algebra for Object-oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*. Springer, 2003.

[4] A. Cavalcanti, A. Sampaio, and J. Woodcock. 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 - FM'03, Pisa, Italy, September 2003*, 2003.

[5] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 2003. To appear.

[6] A. L. C. Cavalcanti and J. C. P. Woodcock. A Weakest Precondition Semantics for Circus. Technical report, University of Kent at Canterbury Computing Laboratory, 2002. In preparation.

[7] B. P. Douglass. *Real Time UML - Developing Efficient Objects for Embedded Systems.* Addison Wesley, 1998.

[8] A. Evans. Reasoning with UML Diagrams. In *Workshop on Industrial Strength Formal Methods, WIFT'98.* IEEE Press, 1998.

[9] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P. Muller, editors, *The Unified Modling Language, UML'98 – Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 336 – 348. Springer-Verlag, 1999.

[10] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java.* PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000.

[11] C. Fischer, E. R. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *LNCS*, pages 91–108. Springer, 2001.

[12] R. Gheyi and P. Borba. Refactoring Alloy Specifications. In *WMF 2003: 6th Workshop on Formal Methods, Brazil*, 2003.

[13] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques.* Technische Universität München, TUM-I9803, 1998.

[14] Martin Gogolla and Mark Richters. On combining semi-formal and formal object specification techniques. In Francesco Parisi-Presice, editor, *Recent trends in algebraic development techniques: 12th international workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997: selected papers*, volume 1376 of *LNCS*. Springer, 1998.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification.* Addison-Wesley, 2nd edition, June 2000.

[16] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[17] D. Harel and O. Kupferman. On Object Systems and Behavioral Inheritance. *IEEE Trans. Software Engineering*, 28(9):889–903, 2002.

[18] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[19] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[20] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[21] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[22] K. Lano and J. Bicarregui. UML Refinement and Abstractoin Transformations. In *ROOM 2 Workshop*, 1998.

[23] B. H. Liskov and J. M. Wing. A Behavioural Notion of Subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6), 1994.

[24] G. Lüttgen, M. Beeck, and R. Cleaveland. A compositional approach to statecharts semantics. In *ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE 2000)*, volume 256 of *ACM Software Engineering Notes*, pages 120–129, San Diego, CA, USA, November 2000. ACM Press.

[25] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall International, 2nd edition, 1997.

[26] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[27] Carroll Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[28] M. Richters and M. Gogolla. OCL: Syntax, semantics, and tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.

[29] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[30] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, 2002.

[31] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 1999.

[32] Rational Software. Rational rose-rt, 2003. URL: www.rational.com/products/rosert.

[33] M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992.

[34] J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[35] J. C. P. Woodcock and J. Davies. *Using Z-Specification, Refinement, and Proof.* Prentice-Hall, 1996.