Using OCL and UML to Specify System Behavior

Shane Sendall and Alfred Strohmeier

Swiss Federal Institute of Technology Lausanne (EPFL)
Department of Computer Science
Software Engineering Laboratory
1015 Lausanne EPFL
Switzerland

email: {Shane.Sendall, Alfred.Strohmeier}@epfl.ch

ABSTRACT Use cases are an excellent tool for capturing behavioral requirements of software systems, but they are not an ideal work product for driving design activities. We believe that there is value from complementing use case descriptions with pre- and postcondition descriptions, not only to better support reasoning about system properties and a basis for testing and debugging, but also to better support a predictable decomposition level on which one can base a systematic transition to design. Nevertheless, we recognize that pre- and postcondition descriptions are not widely used in practice. We believe this is in part due to the formalism used. Either the formalism is too heavy to learn and use, or the formalism does not offer sufficient abstraction from the vocabulary of implementation.

Via an example, the paper highlights our approach for specifying system behavior, which uses the Unified Modeling Language (UML) and its Object Constraint Language (OCL). We focus the paper on pre- and postconditions descriptions and in particular propose a number of enhancements and interpretations to OCL that we made while refining our approach. In particular, we describe a number of issues that cover areas such as the frame problem, incremental descriptions, structuring schemas, and events and exceptions.

KEYWORDS Unified Modeling Language, Object Constraint Language, Pre- and Postcondition Assertions, Software Specification, Requirements Analysis.

1 Introduction

Software development projects are subject to many factors, software and non-software related. It is a balancing act to get the right combination for a given project and its context—there are always trade-offs to make, according to priorities. For example, time-to-delivery and budget typically have a higher priority than rigor of development and quality assurance for most web-based systems [9], where the inverse is normally the case for systems that are human-life critical.

There are more and more projects that are somewhere inbetween safety-critical at the one end and non-critical at the other end. We believe that there is an increasing need for approaches that can provide a reasonable level of quality assurance and rigor in development but still must obey a restrained schedule and budget. For example, at the mid-to-upper range there are many e-business applications, which are “24-7” and mission-critical.

Our goal is to produce an approach for specifying reactive system behavior that can be used in the development of systems that lie in the mid-to-upper range. As a consequence, we have developed a software development method called Fondue [33] that covers the whole software development cycle, uses the UML notations, and is based on the Fusion process [6]. In this paper, we cover an important part of the analysis phase of Fondue.

We have defined a number of criteria that we believe need to be taken into account while developing an approach for the analysis phase:

- The proposed model should be compatible with industry practices and standards.
- The proposed model should be precise so that it can be used as a clear and unambiguous contract for later design activities, as a basis for understanding and documenting the application under development, and to precipitate hidden behavior.
- The proposed model should be targeted towards ease of use for the developer, i.e., it should be simple to learn and use, concise, understandable, etc.
- The proposed model should allow one to manage complexity and size of the description in a modular way that also allows one to localize the effects of change.
- The proposed model should allow one to contain design complexities.
- The proposed model should be conducive to verification activities—via and with the support of tools.
- The proposed model should express "quantifiable" non-functional requirements, such as performance constraints, in an integrated way with the functional requirements.
- The proposed model should be capable of capturing inherent concurrent properties of the system and quality of service properties.

Currently, we believe our approach fulfils, more or less, the first five criteria and goes someway in fulfilling the last three criteria—part of our current and future work.

Our Fondue analysis approach has three principal views:

- a model composed of descriptions of the effects caused by operations, which uses pre- and postcondition assertions written in UML's Object Constraint Language [29], called Operation Schemas [27];
- a model of the allowable temporal ordering of operations, called the System Interface Protocol (SIP) [23]; and
- a model that describes the system state used in the Operation Schemas, called the Analysis Class Model (ACM) [23].

The principal purpose of this paper is the proposition of a number of enhancements and interpretations to OCL that, we believe, make it less laborious to write pre- and postcondition assertions in Operation Schemas and that result in more readable and usable schemas. In particular, we describe a number of issues that cover areas such as the frame problem, incremental descriptions, structuring schemas, and events and exceptions.

The paper is composed of 8 sections. Section 2 provides the motivation for our approach and some justifications for some of the decisions that were made in coming up with Operation Schemas. Section 3 gives a brief overview of Operation Schemas and discusses how OCL is used in schemas. Section 4 presents an elevator case study that is used as an example throughout the paper. Section 5 makes a number of proposals for enhancing and interpreting OCL for use in pre- and postcondition descriptions; there is a continuous thread of proposals throughout. Section 6 discusses some issues related to the work presented and poses some open questions about OCL. Section 7 discusses related work and section 8 draws some conclusions.

2 Motivation for Operation Schemas

The ability of use cases to bridge the gap between the customers and the developers, or more precisely between the non-technical and technical stakeholders, has led to their wide and almost unanimous use in practice. However, use cases are not necessarily the ideal work product for driving design activities due, in part, to their focus on user intentions, which can lead to the unnecessary description of situations that cannot be detected or acted upon by the system. Furthermore, they do not offer sufficient guidelines for obtaining a description with a consistent level of precision, and they are prone to ambiguity and redundancy in their descriptions [17]; consequently they are only supported by tools that are limited to the analysis capabilities of word processors. In addition, use cases do not provide adequate means for dealing with interaction between use cases [13], cannot express state-dependent system behavior adequately [13], and can lead to naïve object-oriented designs in the hands of novice developers if care is not taken [11][12].

One of our first goals was to provide an additional, more precise model to which use cases can be systematically mapped, and which offers fixes to some of the problems encountered with use cases. This proposed model consists of Operation Schemas and a System Interface Protocol (SIP). The mapping process from use cases to Operation Schemas is described in [24]. In short, the use case descriptions are analyzed for events that would trigger system-level operations, these operations are then described using Operation Schemas, and the temporal ordering of those operations is defined in the System Interface Protocol. The advantages of such an approach are the following:

- Consistency of precision is better regulated: the combination of the Analysis Class Model, which defines the vocabulary from which Operation Schemas are defined, and OCL's restricted calculus allow a more consistent level of precision than the natural language descriptions of use cases. Furthermore, iteration between use cases and Operation Schemas focus the developer on refactoring use cases at a more consistent level of precision.
- Use cases are less likely to be over-decomposed (this poses problems, such as, premature design details, and a bias towards functional decomposition designs) because we decompose use cases until we get to the system operation level, and Operation Schemas provide better heuristics on what grain-size a system operation should be.
- Ambiguity is minimized by the calculus of OCL which is based on first-order predicate logic and set theory.
- Redundancy is reduced due to the (decision-making) process of mapping use cases to Operation Schemas.
- Operation schemas and the System Interface Protocol provide a precise means to deal with feature/service interaction (use case interaction at the use case level) and can express state-dependent system behavior.

Ultimately, we hope it is possible to show that the value added by mapping use cases to Operation Schemas is of greater value than the time spent in addition to produce the Operation Schemas.

To improve our chances of achieving this goal we put particular focus on making the Operation Schemas concise and precise, yet simple and easy to use for developers. We found that declarative pre- and postcondition descriptions were a good choice for achieving the first two points, because they offer the ability to specify the essential problem by focusing on what functionality is required—the abstract responsibilities provided/required—rather than its realization. The advantage is that one can abstract above the detail of how the operation is realized in terms of object collaborations for an object-oriented system, for example. Provision for the latter two points was made by enhancing the chosen language for writing pre- and postconditions, i.e. OCL, with a procedural programming language-like style. Also, we made the observation that procedural programming languages are more commonly used in practice compared to declarative ones. We concluded that most developers would therefore be more familiar with this style, as opposed to a declarative style. Our idea was therefore to experiment with a procedural programming language-like facade for Operation Schemas and OCL, and also to provide practical guidelines to some grey areas of pre- and postcondition descriptions such as the frame problem, incremental descriptions, concurrency, etc.

We chose OCL as our formalism for writing Operation Schemas because (1) as we are committed to using UML, OCL is an obvious choice for writing constraints on UML models; (2) OCL already had a operational style that is one step towards what we imagined a procedural style facade in a declarative language to be; and (3) OCL is relatively easy to learn and use, even though we admit that it is sometimes verbose, due to its simple navigation style for constructing constraints—everything, more or less, is achieved by working with and manipulating sets, bags and sequences. The downside that we face with taking OCL on board for Operation Schemas is that it is still a young language and there are therefore still a few unresolved questions on its semantics [22]. Tool support for OCL is progressing and becoming more common (see [31] for a full list of tools), although we admit that major CASE tool vendors still have not shown a lot of interest in OCL, even though it is part of the UML standard [32].

3 Operation Schemas and OCL

An Operation Schema declaratively describes the effect of the operation on an abstract state representation of the system and by events sent to the outside world. It describes the *assumed* initial state by a precondition, and the required change in system state after the execution of the operation by a postcondition, both written in UML's OCL formalism. Moreover, we use the same correctness interpretation as the Larch family of specification languages [14]: when the precondition is satisfied, the operation must terminate in a state that satisfies the postcondition. Operation schemas as we define them here specify operations that are assumed to be executed atomically and instantaneously, hence no interference is possible.

The system model is reactive in nature and all communications with the environment are achieved by asynchronous input/output events. All system operations are triggered by input events, usually of the same name as the triggered operation.

The change of state resulting from an operation's execution is described in terms of objects, attributes and association links, which conform to the constraints imposed by the Analysis Class Model of the respective system. The postcondition of the system operation can assert that objects are created, attribute values are changed, association links are added or removed, and certain events are sent to outside actors. The association links between objects act like a network, guaranteeing that one can navigate to any state information that is used by an operation.

The Analysis Class Model is used to describe all the concepts and relationships in the system, and all actors that are present in the environment, and thus should not be confused with a design class model. Classes and associations model concepts of the problem domain, not software components. Analysis objects do not have behavior and are more closely related to entities from Entity-Relationship models [5] than to design objects.

The standard template for an Operation Schema is shown in figure 1. The various subsections of the schema were defined by the authors, and are not part of the OCL. However, all expressions are written in OCL and conform to our proposals presented in section 5. Each clause is optional except the first. **Pre** and **Post** clauses that are not included default to true and an omitted **Scope** clause defaults to the operation's context, which is the system. The **Declares** clause allows all declarations to be made in a separate and single place, which is in line with the proposal of Cook et al. [7], in contrast to standard use of the *let* construct in OCL, which form part of the expression. A more detailed description of the grammar and usage of Operation Schemas can be found in [27][34].

Operation: This clause displays the entity that services the operation (aka the name of the system of focus), followed by the name of the operation and parameter list.
Description:A concise natural language description of the purpose and effects of the operation.
Notes: This clause provides additional comments.
Use Cases: This clause provides cross-references to related use case(s).
Scope: All classes, and associations from the class model of the system defining the name space of the operation. (Note that it would be possible to have a tool generate this clause automatically from the contents of the other clauses.)
Declares: This clause provides two kinds of declarations: aliasing, and naming.
Aliases are name substitutions that override precedence rules, i.e., treated as an atom, and not just as a macro expansion.
A name declaration designates an object to be "created" by the operation, i.e. the postcondition will state oclIsNew() for it. Each name declares a distinct object.
Sends: This clause contains three subclauses: **Type**, **Occurrence**, and **Order**. **Type** declares all the events that are output by the operation together with their destinations, the receiving actor classes. **Occurrence** declares event occurrences and collections of event occurrences. **Order** defines the constraints on the order of events output by the operation.
Pre: The condition that must be met for the postcondition to be guaranteed. It is a boolean expression written in OCL, standing for a predicate.
Post: The condition that will be met after the execution of the operation. It is a boolean expression written in OCL, standing for a predicate.

Fig. 1. Operation Schema Format

3.1 Presentation of OCL

OCL is a semi-formal language for writing expressions whose principles are based on set theory and first-order predicate logic. OCL can be used in various ways to add precision to UML models beyond the capabilities of the graphical diagrams. Two common uses of OCL are the definition of constraints on class models and the statement of system invariants. As we will see, it can also be used to define pre- and postconditions for operations.

OCL is a declarative language. An OCL expression has no side effects, i.e. an OCL expression constrains the system by observation rather than prescription. OCL is a typed language; it provides elementary types, like Boolean, Integer, etc., includes collections, like Set, Bag, and Sequence, and has an assortment of predefined operators on these basic types. It also allows user-defined types which can be any type defined in a UML model, in particular classes. OCL uses an object-oriented-like notation to access properties, attributes, and for applying operators.

4 Elevator Control Example

For illustrating our approach and for use as a common example throughout this paper, we describe an elevator control system, adapted from [24]. The system controls multiple lifts that all service the same floors of a building. There is a button to go up and one to go down on each floor, which are used to request a lift. Inside each cabin, there is a series of buttons, one for each floor. The arrival of the cabin at a floor is detected by a sensor. The system may ask a cabin to go up, go down or stop. In this example, we assume that a cabin's braking distance is negligible (or that at least the action of stopping the cabin is harmonized with the signal from the floor sensor). The system may ask a cabin to open its door, and it receives a notification when the door is closed; the door closes automatically after a predefined amount of time, when no more people get on or off at a floor. However, neither the automatic closing of an elevator door nor the protection associated with the door closing, stopping it from squashing people, are part of the system to realize.

A scenario for John using the elevator could be: John calls the lift from the 5th floor, choosing to go up. An available lift comes from the 10th floor to the 5th floor to pick him/her up and stops and opens its door. The user gets in and requests the 20th floor. The lift closes its door and goes to the 20th floor. Once it arrives it stops and opens its door. John leaves the lift. A use case that encompasses all scenarios related to a user using the elevator to go from one floor to another can be found in [24].

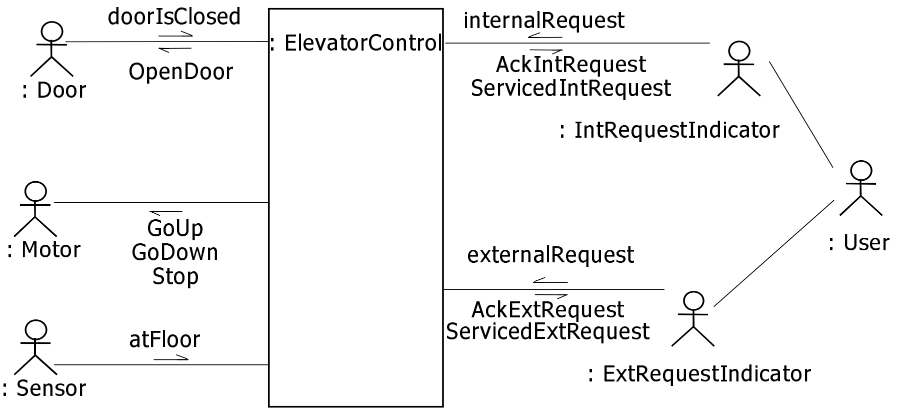

Fig. 2. Collaboration diagram summarizing the interaction between the system and its actors

The system operations for the elevator control system are derived from use case descriptions of the system. How this mapping activity is achieved is not discussed in this paper; interested readers are referred to [24]. The result of this mapping activity from a use case that describes a user taking the lift from one floor to another is shown in figure 2. A (specification-level) collaboration diagram shows four different input events: externalRequest, internalRequest, doorIsClosed, and atFloor, and eight different types of output events: AckExtRequest, AckIntRequest, ServicedExtRequest, ServicedIntRequest, OpenDoor, GoUp, GoDown, and Stop.

The diagram also shows that there is some form of communication between the User actor type and the external request indicator (ExtRequestIndicator) and internal request indicator (IntRequestIndicator) to clarify that the requests originally come from the user. Although we admit this may not be valid UML, strictly speaking, we think showing external communications paths often clarifies the overall working of a system and the consistent exchange of events in the system context.

One could imagine that the indicators control button lights to highlight a pending request.

The Analysis Class Model for the elevator control system is shown in figure 3. It shows all the concepts and relationships between them, the combination of which provide an abstract model of the state space of the system. Inside the system there are five classes, Cabin, Floor, Request, IntRequest, and ExtRequest, and outside six actor classes, Motor, Door, IntRequestIndicator, ExtRequestIndicator, User, and Sensor. The system has five associations: IsFoundAt links a cabin to its current floor, HasIntRequest links a set of internal requests to a particular cabin, HasCurrentRequest links a cabin to its current request, HasExtRequest links the set of all external requests issued by users to the system, and HasTargetFloor links requests to their target floor (source of call or destination). Finally, an <<id>> stereotyped association means that the system can identify an actor starting from an object belonging to the system, e.g., given a Cabin, cab, we can find its corresponding motor via the HasMotor association, denoted in OCL by cab.movedBy. The reason for the <<id>> stereotyped association is that the system can only send an event to an actor that can be identified. Identifying an external actor from inside the system strictly requires an <<id>> stereotyped association.

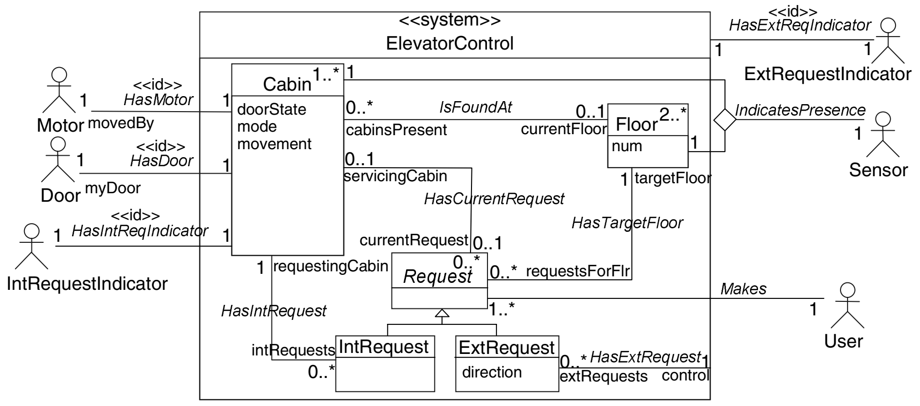

Fig. 3. Analysis Class Model of the Elevator Control System

The System Interface Protocol (SIP) defines the temporal ordering of system operations. An SIP is described with a UML state diagram. A transition in the SIP is triggered by an input event only if the SIP is in a state to receive it, i.e., there exists an arc with the name of the input event. If not, the input event that would otherwise trigger the operation is ignored. A transition from one state to another that has an event as label indicates the execution of the system operation with the same name as the input event.

The Elevator Control SIP is shown in figure 4. It consists of two parallel sub-states. The top-most sub-state models the activity of processing external requests. The dashed line shows that it works in parallel with the lift activities. The Lift submachine, the bottom-most state, is an auto-concurrent statemachine, indicated by a multiplicity of many ('*') in the upper right hand corner. There is a statemachine for each lift[1] but their number is not predefined, hence the multiplicity many. A Lift submachine consists itself of two parallel submachines. The submachine, on the left, models the activity of processing internal requests for the lift. The submachine, on the right, models the functioning of the lift cabin itself.

1. We use the term lift to mean the cabin and it facilities.

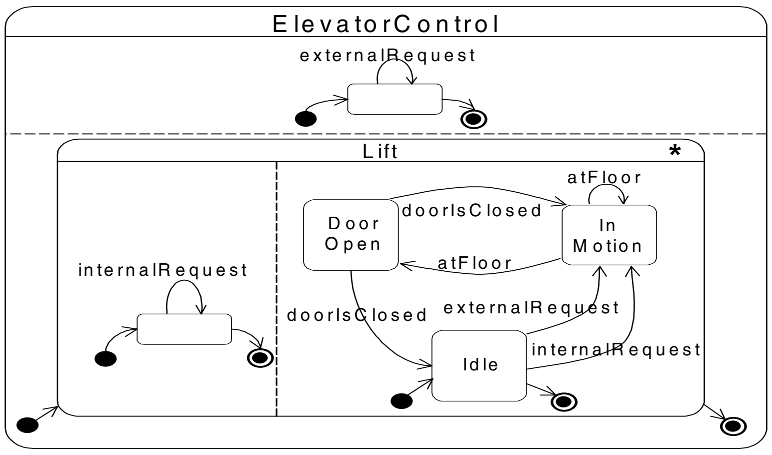

Fig. 4. Elevator Control System Interface Protocol

Each system operation, externalRequest, internalRequest, atFloor, and doorIsClosed, are described by Operation Schemas. However for reasons of size, we highlight just the atFloor Operation Schema, shown in figure 5. The atFloor Operation Schema describes the atFloor system operation. The atFloor system operation occurs as a consequence of a floor sensor detecting the arrival of an elevator cabin at a floor. The system must decide at this point whether there are any requests for the floor that it should service (this will depend on its mode); if so, it will drop off and/or pick up the requesting user(s), otherwise the system will let the lift continue.

The dot notation of OCL usually indicates the traversal of an association, in which case the result is a collection of objects, or the traversal to a property, in which case the result is value of the property. When navigating on association links, the dot notation is used together with the role name, e.g. cab.currentFloor. If there is no explicit role name, then the name of the target class is used as an implicit role name. For example, self.cabin denotes the set of cabins that can be reached by navigating from self, denoting the system instance, on the composition association between the system and the class Cabin.

The arrow operator is used only on collections, in postfix style. The operator following the arrow is applied to the previous "term". For instance, cab.intRequests->select (r | r.targetFloor = f) results in a set consisting of all internal requests r of the cabin, cab, that have the floor f as destination. Note also that we make use of the fact that an IntRequest inherits all the associations of its parent. For instance, it will inherit the association HasTargetFloor that links it to a Floor.

The **Declares** clause defines four aliases that are used for reasons of reuse and to make the postcondition less cluttered. The fourth alias, makeStop, (when substituted) results in true if there is an internal request and/or external request for the supplied floor f that should be serviced by the cabin. The second, third and fourth alias make use of the other aliases and the first alias uses a function calls, detailed in section 5.6.

```
Operation: ElevatorControl::atFloor (cab: Cabin, f: Floor);
Description: The cabin has reached a particular floor, it may continue or stop depending on its
destination and the requests for this floor;
Notes: The system can receive many atFloor events at any one time, each for a different cabin;
Use Cases: take lift;
Scope: Cabin; Floor; Request; IntRequest; ExtRequest; HasIntRequest; HasExtRequest;
HasCurrentRequest; HasTargetFloor; IsFoundAt;
Declares:
  reqsToStopFor: Set (Request) Is
                    calcAllowedStops (cab, f, cab.intRequests->select (r | r.targetFloor = f),
                                      self.extRequests->select (r | r.targetFloor = f));
  pickUpRequest: Set (ExtRequest) Is reqsToStopFor->select (r | r.oclIsType(ExtRequest));
  dropOffRequest: Set (IntRequest) Is reqsToStopFor->select (r | r.oclIsType(IntRequest));
  makeStop: Boolean Is reqsToStopFor->notEmpty ();
Sends:
  Type: Motor::{Stop;}; Door::{OpenDoor;};
      ExtRequestIndicator::{ServicedExtRequest;}; IntRequestIndicator::{ServicedIntRequest;};
  Occurrence: stop: Stop; open: OpenDoor;
  Order: <stop, open>; -- the output events are delivered in the order "stop followed by open"
Pre:
  cab.movement <> Movement::stopped; -- cab was moving
Post:
  cab.currentFloor = f & -- new current floor for the cabin
  if makeStop then -- someone to drop off or pick up
      (cab.movedBy).sent (stop) & -- stop sent to cab motor
      cab.movement = Movement::stopped &
      (cab.myDoor).sent (open) & -- open sent to door
      cab.doorState = DoorState::open &
      self.request->excludesAll (reqsToStopFor) & -- removed all serviceable requests for this floor
      if pickUpRequest->notEmpty () then
          (self.extReqIndicator).sent (ServicedExtRequest (
              (callingFlr => pickUpRequest->any (true).targetFloor,
               dir => pickRequest->any (true).direction)))
      endif &
      if dropOffRequest->notEmpty () then
          (self.intReqIndicator).sent (ServicedIntRequest (
              (destFlr => dropOffRequest->any (true).targetFloor))) --inform int. request is serviced
      endif
  endif;
```

Fig. 5. atFloor Operation Schema for the Elevator Control System

The **Sends** clause shows that instances of the event types Stop, OpenDoor, ServicedExtRequest, ServicedIntRequest may be sent to the indicated actors (**Type** subclause) and that Stop and OpenDoor have named instances (**Occurrence** subclause). It also defines a sequencing constraint on the output events that states that the two event instances are delivered to their respective actors in the order "stop followed by open" (**Order** subclause). The **Pre** clause states that the cabin, cab, is currently moving.

The first line of the **Post** clause states that the cabin is now found at floor f with the isFoundAt association updated accordingly. The next (compound) expression states that if the lift has a request for this floor, then the cabin's motor was told to stop, the cabin's door was told to open, the state attributes of the cabin were updated, and the requests that were serviced by this stop were removed from the system state.

The expression, self.request->excludesAll (reqsToStopFor), not only removes the serviced request objects from the system (discussed in section 6.2), but deletes also all the association links connected to the deleted objects. This sort of implicit removal ensures consistency of associations and is explained in section 5.1.

Also, the & operator used throughout the schema is a shorthand for logical "and" it is discussed in section 5.4. In the **Post** clause, we assert that an actor is sent an event using the "sent" shorthand, which indicates that the supplied event instance was placed in the event queue of the appropriate actor instance; this is detailed in section 5.7. Looking further at the OCL notation, an expression, such as cab.doorState = DoorState::open, means that the attribute, doorState, of the object cab has the enumeration value open (of the type DoorState) after the execution of the operation.

5 Proposals

In this section, we make a number of proposals to OCL. It has a number of subsections, but there is a continuous thread of proposals for enhancements and interpretations to OCL throughout. We cover such areas as the association consistency assumption, the frame problem, incremental descriptions, structuring schemas, and events and exceptions.

5.1 Consistency of Associations

An association link can only link existing objects; it is therefore a well known consistency constraint for class models that when an object is removed from the system state all association links connected to it have to be removed too. Although it would be possible to explicitly state all association links that must be destroyed, this is quite cumbersome in the presence of numerous associations. Therefore we propose the association consistency assumption.

Assumption 1: Removal of an object from the system implies implicitly that all association links in the system that included the destroyed object are destroyed, in addition.

5.2 Frame Assumption

The frame of the specification is the list of all variables that can be changed by the operation [18], which in our model is always a subset of all objects and all associations links that are part of the system state. The postcondition of a specification describes all the changes to the frame variables, and since the specification is declarative, the postcondition must also state all the frame variables that stay unchanged. The reason is simple: if the unchanged frame variables are left unmentioned, they are free to be given any value and the result will still conform to the specification.

Formal approaches such as Z, VDM, Larch, etc. explicitly state what happens to each one of these frame variables—even for those variables that stay unchanged. This approach soon becomes cumbersome to write and error-prone, particularly for specifications that have complex case distinctions (where the complete frame is the combination of all the variables read/changed in each different case). One approach that avoids this extra work is to imply a "... and nothing else changes" rule when dealing with these types of declarative specifications [3]. This means that the specification implies that the frame variables are changed according to the postcondition with the unmentioned frame variables being left unchanged. This approach reduces the size of the specification, thus increases its readability, and makes the activity of writing specifications less error prone. We therefore adhere to this convention.

However, there is a slight problem with this assumption in the case of implicit removal—a consequence of the association consistency assumption. For an example, let us reconsider the seventh line of the postcondition in figure 5.

```
self.request->excludesAll (reqsToStopFor)
```

If we strictly apply the frame assumption "... and nothing else changes", as a result the associations HasIntRequest, HasExtRequest, HasCurrentRequest, and HasTargetFloor would stay unchanged which would lead to an inconsistent system state. At least three of the associations have to be changed, and will be changed following our implicit consistency of associations convention stated in section 5.1.

Also, we need to cover two more cases: what happens to attributes of frame objects that are not mentioned by the postcondition, and what happens to attributes of newly "created" objects that are not mentioned in the postcondition.

We propose the following amended frame assumption.

Assumption 2: No frame variables (including, if a variable denotes an object, the object attributes) are changed with the execution of the operation other than those that are explicitly mentioned to be changed by the postcondition, the associations that are implicitly modified as defined by the association consistency assumption, and the objects, and their attributes, that are new to the system state as a consequence of the operation.

This assumption forces all attributes of objects that are not mentioned to keep the same value with the exception of new objects added to the system state; in this case, we provide three possible interpretations: 1) attributes of new objects that are not mentioned in the postcondition can take any value, 2) the unmentioned attributes get predefined default values, or 3) the specification is incorrect if a value is not given to the respective attributes. The last interpretation gives more of a prescriptive flavor and one could probably expand this to also prohibit specifications where attribute values are constrained to a range rather than a precise value, e.g., acc.num > 0 would not be allowed in the description of an effect.

5.3 Incremental Descriptions

It is common practice in software development to tackle a problem in a piece-meal fashion—describing the problem incrementally. In a similar way, it is useful to describe postconditions incrementally, i.e., a particular effect of an operation may be defined by a combination of constraints that are defined at different places throughout the postcondition. However, declarative specifications do not in general support incremental descriptions. For example, it is *not* possible to state the effect on a given set in the following way (an extract of an *inconsistent* postcondition),

```
req.targetFloor = req.targetFloor@pre->including (flr1)
...
req.targetFloor = req.targetFloor@pre->excluding (flr2)
```

because both conditions define a different final set, but nevertheless refer to the same one: clearly a contradiction.

Set manipulations, like the two from above, are commonplace in OCL, and there are many reasons, detailed later, why an incremental description of sets is advantageous. For this purpose, we introduce the principle of minimum set into OCL to facilitate incremental description of effects on sets in postconditions.

5.3.1 Minimum Set Principle

Proposal 1: The minimum set principle is applied to the interpretation of postconditions.

We propose to define the semantics of OCL postconditions by applying the principle of minimum sets. For each class and each association within the system, we will consider their sets of instances and links, and claim that these are all minimum sets after execution of the operation.

Unless otherwise stated, if C is a class of the system, if SetOfAllObjects(C)@pre is the set of its instances before the execution of the operation, and SetOfAllObjects(C) is the set of its instances after the execution of the operation, then SetOfAllObjects(C) is the minimum set containing SetOfAllObjects(C)@pre and fulfilling the postcondition. Intuitively, SetOfAllObjects(C) can be constructed by adding to SetOfAllObjects(C)@pre all instances of C created by the operation. Similarly, we can come to a similar result for an association A of the system, where SetOfAllLinks(A) is the minimum set containing SetOfAllLinks(A)@pre and fulfilling the postcondition. The rule must hold for all classes and associations. Also, the minimum set principle is quite complementary to our frame assumption that states, more or less, that nothing changes other than what is explicit in the postcondition.

There is a slight problem, however, when we allow for the destruction of objects or removal of association links. For defining the semantics, the idea is then to gather the deleted entities into a temporary set, and rephrase the rule in the following way: let A be an association of the system, let us denote by SetOfLinksToRemove(A) the set of links of A destroyed by the operation, then SetOfAllLinks(A) ∩ SetOfLinksToRemove(A) is empty, and SetOfAllLinks(A) ∪ SetOfLinksToRemove(A) is equal to SetOfAllLinks(A)@pre, where SetOfAllLinks(A) is the minimum set. The minimum set principle has a similar effect to re-dashing of schemas when composing them in Z [21].

Applying the minimum set principle to a postcondition, for example, we could rewrite the condition,

```
req.targetFloor = req.targetFloor@pre->including (flr)
```

as:

```
req.targetFloor->includes (flr)
```

which would be equivalent as long as no other effects have been expressed about the state of req.targetFloor.

One consequence of the minimum set principle that is not so intuitive is the case where the condition is negated. For example, the following condition:

```
not req.targetFloor->includes (flr)
```

is *not* equivalent to the condition:

```
req.targetFloor->excludes (flr)
```

because the first condition states that flr is not one of the elements added to the set req.targetFloor, where the second condition states that flr was one of the elements to be removed from the set req.targetFloor@pre.

The minimum set principle can also be applied to collections in general.

However, when it is applied to a bag, duplicates are not accounted for, e.g.,

```
Pre:   bagX = Bag {};
Post:  bagX->includes (x1) and
       bagX->includes (x1);
```

is equivalent to:

```
Post:  bagX->includes (x1);
```

An additional constraint it therefore required for the bag to contain two x1 elements, e.g., bagX->count(x1) = 2.

We take the opportunity, even though this has nothing to do with the minimum set principle, to insist that ordering of conditions does not suffice to order elements in a sequence, e.g.,

```
Pre:   seqX = Sequence {};
Post:  seqX->includes (x1) and
       seqX->includes (x2);
```

does not mean that x1 precedes x2 in seqX. The correct postcondition would be

```
seqX = seqX@pre->union (Seq{x1, x2})
```

which, using the minimum set principle, can be simplified to:

```
seqX->union (Seq{x1, x2})
```

The minimum set principle allows one to write the postconditions incrementally. For example, we could define a fragment of the postcondition for an imagined operation called swapCabins, which exchanges two cabins, cab1 and cab2, from floors f1 to f2 and from f2 to f1, respectively:

```
f1.cabinsPresent->excludes (cab1) and
f2.cabinsPresent->includes (cab1) and
f2.cabinsPresent->excludes (cab2) and
f1.cabinsPresent->includes (cab2)
```

Taking this example further we could be later asked, in a maintenance phase for example, to modify the operation by putting the cabin specialCab at the floor f1, as part of the swapCabins operation. This would simply require the following additional line in the postcondition.

```
... and
f1.cabinsPresent->includes (specialCab)
```

Such an incremental description, possible because of the minimum set principle, is easier to maintain because conditions are not definitive can be extended constructively.

```
if inPriorityMode then
        self.extRequest->includes (r1)
else
        self.extRequest->excludes (r5)
endif and
if liftAtSameFloor then
        self.extRequest->excludes (r2)
else
        self.extRequest->includes (r4)
endif and
if liftIsBusy then
        self.extRequest->includes (r6)
else
        self.extRequest->excludes (r3)
endif
```

Fig. 6. Incremental Approach

Incremental descriptions also offer one the possibility to break large case distinctions into more manageable and concise *if-then-else* conditions. Normally with a non-incremental description, one would specify each individual case completely with possibly many repetitions. Figure 6 shows an example of an incremental description using three *if-then-else* conditions. Figure 7 shows an equivalent description that is not incremental, it uses eight implication conditions.

Note that the relationship between the two approaches is exponential: there are 2^n separate cases for *n* if-then-else conditions. Note that we used implications in figure 7, but it is common practice to use separate pre/post pairs (see section 5.6 for more details), which would make the text even longer. Clearly, the usefulness of our incremental descriptions becomes even clearer as *n* gets larger.

```
inPriorityMode and liftAtSameFloor and liftIsBusy implies
        self.extRequest = self.extRequest@pre->union (Set {r1,r6})->excluding (r2) and
inPriorityMode and not liftAtSameFloor and not liftIsBusy implies
        self.extRequest = self.extRequest@pre->union (Set {r1, r4})->excluding (r3) and
inPriorityMode and not liftAtSameFloor and liftIsBusy implies
        self.extRequest = self.extRequest@pre->union (Set {r1, r4, r6}) and
inPriorityMode and liftAtSameFloor and not liftIsBusy implies
        self.extRequest = self.extRequest@pre->including (r1) - Set {r2, r3} and
not inPriorityMode and liftAtSameFloor and liftIsBusy implies
        self.extRequest = self.extRequest@pre->including (r6) - Set {r5, r2} and
not inPriorityMode and not liftAtSameFloor and not liftIsBusy implies
        self.extRequest = self.extRequest@pre->including (r4) - Set {r5, r3} and
not inPriorityMode and not liftAtSameFloor and liftIsBusy implies
        self.extRequest = self.extRequest@pre->union (Set {r4, r6})->excluding (r5) and
not inPriorityMode and liftAtSameFloor and not liftIsBusy implies
        self.extRequest = self.extRequest@pre - Set {r5, r2, r3}
```

Fig. 7. Non-Incremental Approach

5.3.2 Incremental Plus and Minus

Continuing with incremental descriptions, we can use an idea similar to the minimum set principle for numeric types. We propose to use the operators, "+=" and "-=" with the following meaning: the value of the numeric entity in the post-state is equivalent to the value in the pre-state plus all the right-hand sides of all += operators used in the post-condition that refer to the numeric entity, and minus all the right-hand sides of all -= operators that refer to the numeric entity. For example:

```
obj.x += 5 and
obj.x -= 4
```

is equivalent to (and can be rewritten as):

```
obj.x = obj.x@pre + 1
```

However, care needs to be taken when the incremental style is mixed with the other styles.

```
Post: ...
obj.x += 5 and              -- line one
obj.x -= 4 and              -- line two
obj.x = obj.x@pre + 2 and   -- line three
obj.x = 2                   -- line four
```

The above example is an erroneous specification: line three is in contradiction with the result defined by the incremental plus and minus, and line four would require that obj.x@pre be either 0 or 1 depending on whether line three was brought into agreement with line one and two or vice versa.

Unfortunately, the incremental plus and minus facility cannot be extended to more complex expressions (e.g. multiplication) because it relies on the commutativity of additions and subtractions.

5.4 Shorthand Proposals for OCL

In this subsection, we propose some shorthand notations that could be used in OCL to make the job of the specifier less laborious and therefore probably less error-prone.

Proposal 2: The *else* part in the *if-then-else* construct is optional if the resulting type of the *if-then-else* expression is boolean.

We use *if-then-else* conditions for case distinction. Without our frame assumption, we would write:

```
if makeStop then
    cab.movement = Movement::stopped
else
    cab.movement = cab.movement@pre
endif
```

Taking the frame assumption into account, we are not required to state which variables stay unchanged; consequently, the *else* part, in this case, is simply true:

```
if makeStop then
    cab.movement = Movement::stopped
else
    true
endif
```

Applying proposal 2, the *else* part of the *if-then-else* condition can be made implicit:

```
if makeStop then
    cab.movement = Movement::stopped
endif
```

An implicit else true promotes a smaller, more readable postcondition. Thus, every time that a postcondition includes an *if-then* statement (without the *else* part), the expression is of type boolean and an else true is implied. To support this feature in OCL, only a syntax change would be required.

Proposal 3: *elsif* parts can be used in if-then-else constructs.

When dealing with a large number of branches by using the *if-then-else* construct, it is common to nest *if-then-else* constructs within other *if-then-else* constructs. This practice, however, can become problematic as the depth of nesting increases, having a negative impact on clarity for the reader and even the writer. Using an *elsif* part for the *if-then-else* construct can help make such situations cleaner and clearer to write and understand. The *elsif* addition is directly derivable from nested *if-then-else* constructs. For example, the following two conditions are equivalent:

```
if condA then                |  if condA then
        effectA              |          effectA
elsif condB then             |  else
        effectB              |          if condB then
else                         |              effectB
        effectC              |          else
endif                        |              effectC
                             |          endif
                             |  endif
```

The required change to OCL to realize this proposal would be purely a syntactic one, i.e., the *elsif* keyword is simply added as a new part of the *if-then-else* construct. Furthermore, one could treat the new construct as a syntactic rewrite of the *if-then-else* construct.

Proposal 4: The commercial-and "&" can be used for separating conjunctive effects. It is an alternative to repeating the "pre" and "post" keywords.

OCL uses the keywords “pre” and “post” to separate system effects in a pre- and post-condition respectively [32]. However, we believe this becomes quite heavy and disruptive for the reader of large specifications. Therefore, we propose to use the commercial-and “&” as an effect description separator, because it is more discrete for use in large specifications and it has a clear usage with *if-then-else* conditions.

The commercial-and “&” and the logical-and “**and**” operators are logically equivalent. The difference between them lies in their interpretation that helps the human reader. Commercial-and, &, is used to separate conjunctive system effects, whereas logical-and, **and**, is used to separate boolean expressions that form conditions for branches leading to different effects.

For example, the following two postconditions are equivalent, but the one on the left-hand side is written using the multiple post keywords style, and the one on the right-hand side is written using our proposed style.

```
Post:  effectA                        | Post:  effectA &
Post:  if condA and condB then        |        if condA and condB then
           effectB                    |            effectB &
           and effectC                |            effectC
       endif                          |        endif &
Post:  effectD                        |        effectD &
Post:  effectE                        |        effectE
```

Again the change required to OCL to accommodate this proposal would be simply a syntactic one.

5.5 Interpreting Composite Values in OCL

Composite values for objects and events (section 5.7) are very useful in comparing and matching values between entities.

Proposal 5: The aggregate notation can be used for denoting composite values.

We propose the optional use of an Ada-style aggregate notation for denoting composite values. The value attributes of an object, and the parameters of an event correspond to composite values.

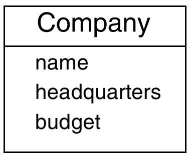

Fig. 8. Company Class in UML class notation

An aggregate is written by associating with each attribute a value, denoted by its name. The following aggregate conforms to figure 8:

```
(name => “Microsoft”, headquarters => “Richmond”, budget => 50.0E9)
```

Positional notation is also possible, but then the ordering of the values must be agreed upon by some convention, e.g. alphabetical order of the attribute names for objects:

```
(50.0E9, “Richmond”, “Microsoft”)
```

For clarity, it is sometimes useful to be able to qualify an aggregate by its type, e.g. the class name or the event type, yielding a so-called qualified aggregate. We propose to use the Ada-like “tick” notation, i.e. the type name precedes the aggregate, separated by an apostrophe, e.g.

```
Company’(50.0E9, “Richmond”, “Microsoft”)
```

Due to the “by-reference” semantics of objects and events, we propose to denote their composite value by introducing the property “all”. Thus we can write expressions like the following:

```
company.all = (name => “Microsoft”, headquarters => “Richmond”, budget => 50.0E9)
```

which evaluates to true if the object referenced to by company has the corresponding attribute values. The above condition is equivalent to:

```
company.name =“Microsoft” and
company.headquarters = “Richmond” and
company.budget = 50.0E9
```

The advantage of aggregates is that related values are kept together in one place.

Proposal 6: The class or event name together with an aggregate can be used to denotate a representative object or event.

We introduce a special shorthand that makes it possible to match objects and events directly to composite values. The shorthand is defined for each object/event type. It uses the same name as the type, and it takes a composite value as parameter, resulting in a reference to the corresponding object/event in the system that has the matching composite value. For example, the expression:

```
Company ((50.0E9, “Richmond”, “Microsoft”))
```

results in all the company objects that have the corresponding composite value and chooses one, if there are more than one.

The above expression is a shorthand for the following expression:

```
Company.allInstances->select (c | c.all = (50.0E9, “Richmond”, “Microsoft”))->any (true)
```

The precondition of the any collection operator states that the supplied collection, i.e., the expression on the left-hand side, must have at least one element satisfying the expression. This means that if there are no objects matched, then the shorthand is undefined. Thus, the specifier should ensure that the corresponding object exists for all valid system states.

Such a shorthand allows one to write concise and, we believe, intuitive expressions in postconditions, e.g.:

```
region.localCompanies->includes (Company ((50.0E9, “Richmond”, “Microsoft”)))
```

which results in true if Microsoft is a member of the local companies in the region, region. This shorthand notation is particularly useful for denoting event sending, as we will see in section 5.7.

Proposal 7: The oclIsNew property can be optionally parameterized by a composite value, which states all the (initial) attribute values of the object.

We propose to allow the oclIsNew property to be parameterized with a composite value, defining the attribute values of the new object. For example, asserting that a new object has the same value as another one can be described simply by:

```
myCompany.oclIsNew (john.company.all)
```

It is also possible to use an aggregate, which denotes the actual attribute values. For example, a postcondition could state:

```
cabinX.oclIsNew ((doorState => DoorState::closed, mode => Mode::express,
                  movement => Movement::stopped))
```

which means that the object, cabinX, became a new element of the system state with the execution of the operation, and all its value attributes, i.e., doorState, mode, and movement, were given the enumeration values, closed, express, and stopped, respectively.

The above expression is directly equivalent to the following one:

```
cabinX.oclIsNew and
cabinX.doorState = DoorState::closed and
cabinX.mode = Mode::express and
cabinX.movement = Movement::stopped
```

The proposed notation ensures that all attributes of a newly created object were constrained to the given values, and none of them were forgotten.

Proposal 8: The oclIsNew property can be applied to a collection; it then takes a single parameter which signifies the number of new objects that were created with the execution of the operation and are the only members of the collection.

We propose to allow the creation of a collection of objects by introducing the oclIsNew property for collections, where the oclIsNew property takes as parameter the number of elements to be created. Assuming colX: Collection (X), then:

```
colX.oclIsNew (n)
```

is equivalent to,

```
colX->forall (x: X | x.oclIsNew) and
colX->size () = n
```

Both conditions state that the collection contains exactly n new objects. For example, a postcondition defining the result of initializing the ElevatorControl system could include the following extract:

```
self.cabin.oclIsNew (5)
```

which is equivalent to the following condition:

```
self.cabin->forall (c: Cabin| c.oclIsNew) and
self.cabin->size () = 5
```

5.6 Structuring Schemas by Parameterized Predicates and Functions

For the sake of readability and usability, it is necessary to be able to structure Operation Schemas as the size of a schema increases. One common approach is to use multiple pre- and postcondition pairs for structuring operations (called case analysis in the Larch community) [30][8], where each schema describes the effect by the operation in a distinct case. We avoid this style of structuring because it leads to the case explosion problem that was demonstrated by figure 7.

Even though incremental descriptions help reduce the size of a specification, specifications can nevertheless get large and we have made the observation that they become cumbersome to write and use. This observation is backed by the results of a controlled experiment by Finney et al.; the result of the experiment gives evidence that structuring a specification into schemas of about 20 lines significantly improves comprehensibility over a monolithic specification [10].

We introduce two new concepts that help structure Operation Schemas and provide a means for reuse. We call them parameterized predicates and functions.

A *parameterized predicate* can be used in **Pre** and **Post** clauses to better support readability of schemas and to allow one to reuse commonly recurring predicates. They are inspired from those proposed in Catalysis [8]. They are used to encapsulate a 'piece' of the pre- or postcondition and therefore they can use the suffix '@pre' (in the case that it is destined for postconditions); they evaluate to true or false. They implicitly refer to self, the system object of the schema where they are instantiated. At definition, their scope is the schema (i.e. the names declared in the **Scope**, **Declares** and **Sends** clauses) where it is supposed to be used; it can then be used in all schemas having this scope or a wider one. When a predicate is referred to in a postcondition, it must be possible to resolve all references within the current context. For example, we could define a parameterized predicate that encapsulates the condition that the state of a cabin was changed to stopped and open, and two events were sent to the motor and the door to stop and open, respectively:

```
Predicate: madeStop (targetCabin: Cabin, stopTheLift: Stop, openTheDoor: OpenDoor);
Body:      (targetCabin.movedBy).events->includes (stopTheLift) and -- stop sent to cab motor
           targetCabin.movement = Movement::stopped and
           (targetCabin.myDoor).events->includes (openTheDoor) and -- open sent to door
           targetCabin.doorState = DoorState::open;
```

The parameterized predicate can then be used in the postcondition of the atFloor Operation Schema of figure 5, for example, in the following way:

```
Post: ...
    if makeStop then
        madeStop (cab, stop, open) and -- use of parameterized predicate
        self.request->excludesAll (reqsToStopFor) and
        ...
```

A *function* may be used to encapsulate a computation. They do not have any side effects, i.e. they are pure mathematical functions, and to the contrary of a system operation they do not change the system state. Functions may be used as a reuse mechanism for commonly recurring calculations.

We propose to separate the function declaration (its signature) from the function definition. In that way, they can be used as a placeholder when the need for the function is known, but its realization is deferred to a later stage of development, i.e. design or implementation. For example, we might know that we have to determine the best suited lift to service a particular request, which can be expressed by a function, e.g.

```
Function: bestSuitedCabin (options: Set (Cabin), requestedFlr: Floor): Cabin;
-- A function that hides the algorithm for choosing the best suited cabin to service a request
```

but the choice of the algorithm is deferred until design.

Functions can also be used when OCL is not suitable for expressing the algorithm, e.g. in the case of numeric computations. Functions are therefore a way to escape the limited expressive power of OCL when necessary. However, we admit that such a facility can be abused.

Functions can be referred to anywhere, in contrast to parameterized predicates, whose use is limited to pre- and postconditions. They can refer to the model elements of the Analysis Class Model. If a function does not refer to any model elements, then it is a universal function, e.g. the sine function, and it is possible to refer to it "anywhere". Functions can include an **Aliases** clause, which is local to the function, and is equivalent to the **Declares** clause of Operation Schemas, except only aliases are allowed. When referring to a function, it must be possible to resolve all references within the current context.

For example, the first line after the **Declares** of figure 5 (atFloor Operation Schema) used a function calcAllowedRequests, which returns a possibly empty set of requests to service. A possible definition of the function could be:

```
Function: calcAllowedRequests (c: Cabin, currentFlr: Floor, intReqs: Set (IntRequest)
                                    extReqs: Set (ExtRequest)): Set (Request);
Function Body: calcAllowedRequests (c: Cabin, currentFlr: Floor, intReqs: Set (IntRequest)
                                    extReqs: Set (ExtRequest)): Set (Request);
Aliases:
  atFloorExtremities: Boolean Is currentFlr.num = MIN_FLOOR_NUM or
                                    currentFlr.num = MAX_FLOOR_NUM;
Post:
  if intReqs->notEmpty () and
        ( intReqs->any (true) = c.currentRequest or allowedToDropOff (c.mode)) then
    result->includes (intReqs->any (true))
  endif &
  if extReqs->notEmpty () and
        (extReqs->exists (r | r = c.currentRequest) or allowedToPickUp(c.mode)) then
    if extReqs->exists (r | r = c.currentRequest) then
      result->includes (c.currentRequest)
    else -- allowed to make a pick-up
      if atFloorExtremities then
        result->includes (extReqs->any (true))
      else
        result->includes (extReqs->select(r | r.direction = cab.movement)->any (true))
      endif
    endif
  endif;
```

This function calculates the internal and external requests that are allowed to be serviced according to the mode of the lift, and the context, e.g., direction the lift is going, etc.

5.7 Events, Calls, and Exceptions

Operation schemas specify not only the changes to the system state, but also the *system events* that are output by the operation. Communications between the system and actors are through event occurrence delivery. In our approach, we distinguish input from output events. Input events are incoming to the system and trigger system operations. Usually, their names are the same. The parameters of the input event are the parameters of the system operation. Output event occurrences are outgoing from the system and are delivered to a destination actor.

We propose to interpret a system event occurrence as:

- having by-reference semantics;
- having unique identity;
- having an implicit reference to its sender, referred to by the keyword *sender*;
- being reliably and instantaneously delivered (no latency).

There are several kinds of system events, which can be thought of as either a specialization of SignalEvent or CallEvent in UML, depending on whether the event is asynchronous or synchronous. We will distinguish three kinds of system event types that we call Event, Exception, CallWithReturn, respectively stereotyped <<event>>, <<exception>>, and <<callwithreturn>>. They all have a single compartment containing parameters.

An Event occurrence instigates an asynchronous communication; it usually triggers the execution of an operation. An Exception occurrence signals an unusual outcome to the receiver, e.g., an overdraft of an account (section 5.7.2). A CallWithReturn occurrence triggers the synchronous execution of an operation that returns a result to the sender (section 5.7.1). The result is modelled by an Event occurrence.

Often we use the term event with the meaning of any of the above kinds or even occurrences.

We use a naming convention to differentiate the different kinds of events: suffix "_e" for an Exception, and suffix "_r" for CallWithReturn. The reason for this naming convention is to help specifiers visually differentiate between different kinds of events.

Care must be taken that all parameters of an event sent by an operation have defined values.

The System Interface Protocol defines the temporal ordering of the input events (as shown in figure 4 for the elevator control system), but the events that are output by the system during the execution of an operation are specified in the respective schema. This is achieved by stating:

- the type of the event and the destination actor type;
- the condition(s) under which the event occurrence is sent;
- the actual parameters of the event occurrence;
- the destination actor instance(s);
- and optionally any ordering constraints that the event occurrence may have relative to other events output by the same operation.

The declaration of output events is written in the **Sends** clause of the Operation Schema. The **Sends** clause is broken up into three (optional) sub-clauses called **Type**, **Occurrence**, and **Order**. The **Type** sub-clause declares the actor types together with the event types that may be sent. The **Occurrence** sub-clause declares the named event occurrences. The **Order** clause defines the constraints on the order that the events are output.

As an example, let us consider a **Sends** clause of an Operation Schema for a subsystem of the elevator control system called cabin controller, which communicates with the scheduler subsystem, the administration subsystem, the motor, and the door (figure 9).

```
Sends:
    Type:   Motor::{Stop;}; Door::{OpenDoor;}; Administration::{LogMessage;};
            Scheduler::{GetNextRequest_r Throws NoRequests_e;};
    Occurrence: stopLift: Stop; openLiftDoor: OpenDoor; gnr: GetNextRequest_r;
            seqMessages: Sequence (LogMessage);
    Order: <seqMessages, stoplift, openLiftDoor>;
```

Fig. 9. Sends clause of an Operation Schema

It states by the **Type** sub-clause that actor instances of type Motor, Door, Administration, and Scheduler may be sent occurrences of the events Stop, Open, Message, and GetNextRequest_r respectively. It also uses the **Throws** keyword to indicate that an occurrence of the exception NoRequest_e may be received by the operation instead of a reply from the call triggered by an occurrence of GetNextRequest_r.

The **Occurrence** sub-clause declares an event occurrence of type Stop, Open, GetNextRequest_r, and a sequence of event occurrences of type Message. The **Order** sub-clause states that the sequence of message occurrences are sent before the stoplift occurrence, and the stoplift occurrence is sent before the openLiftDoor occurrence.

We propose that declaring a group of events as a sequence means that they are received in the order that they are in the sequence, and events in a set or bag are not ordered.

Moreover, if a collection is specified, ordering is not dealt with, but deferred to later design activities.

All event occurrences have to be created within the execution of the operation. Therefore we propose to avoid explicitly stating that they were created in the postcondition (to the contrary of new objects in the system).

Each actor has an event queue—just as the system has an event queue. If the actor is able to deal with occurrences of a given event (type), then it is possible to state that an event was placed in the actor's (input) event queue as a result of an operation.

Hence, an event is specified as delivered by asserting that it is present in the event queue of the destination actor. For example, the output of a request to the door actor of the cabin controller system can be asserted in the postcondition of the Operation Schema, corresponding to figure 9, in the following way, given that there is an association from the cabin controller to its door that has the role name myDoor.

```
Post:
...
(self.myDoor).events->includes (openLiftDoor)
...
```

In addition to explicitly writing that an event is placed on the target actor's event queue, we propose a shorthand that we have found in practice to be more intuitive to users and writers. It has the following form, where actorX denotes any identifiable actor and eventOccurrenceX denotes any appropriate event occurrence:

actorX.*sent* (eventOccurrenceX)

and is equivalent to or syntactic sugar for:

actorX.events->includes (eventOccurrenceX)

We emphasize that *sent* is just a shorthand and should not be confused with a property of the actor.

Proposal 9: The delivery of an event in an OCL postcondition is asserted by placing the event occurrence in an actor's event queue.

Note that because events have by-reference semantics, an event can be placed in several event queues (multicast).

For example, we could imagine a situation where a fire alarm triggers a system operation that stops all moving lifts. An extract of the postcondition that asserts the output of a stop event to all moving cabins could be the following, given emergencyStop: Stop and movingCabins = self.cabin->select(c | c.movement <> Movement::stopped):

movingCabins.movedBy -> forall (m | m.*sent* (emergencyStop))

5.7.1 Modeling Results Returned by Operations

In this subsection, we discuss our ideas on how to use Operation Schemas for modeling results returned by operations to other actors or subsystems.

Figure 10 shows two approaches for servicing a particular request from an actor. The two approaches produce the same result. The first approach (top) shows a blocking call from requestingActor to subsystemA. During the execution of this operation, subsystemA executes a blocking call to subsystemB. Once the call returns, subsystemA returns the result of the request to requestingActor. For modeling this situation, we will use CallWithReturn occurrences and operations returning results.

The second approach (bottom) achieves the same result by exchanging asynchronous events. Consequently, two asynchronous calls are made to subsystemA, as opposed to a single synchronous call in the first approach. This second case is handled with sending event occurrences as we have already seen in this paper. It is our preferred approach and we recommend it for systems specified from scratch.

However, both approaches are needed when we are modeling already existing components.

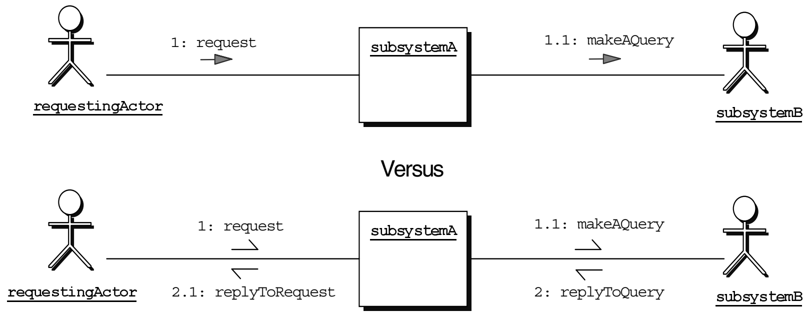

Fig. 10. Alternatives for Returning Results from "Calls"

A CallWithReturn occurrence has an associated result event (figure 11). It is possible to navigate to this returned result.

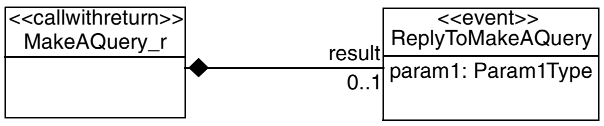

Fig. 11. Relationship between a CallWithReturn and its Reply

With the event declarations shown in figure 11, here is a postcondition fragment that asserts that a CallWithReturn occurrence was delivered to subsystemB and shows how the returned result can be accessed via result.

```
subsystemB.sent (makeAQuery) & -- like for a non-blocking call
objX.addr = makeAQuery.result.param1 -- note the reply event has possibly many return parameters
```

The first line asserts that the event makeAQuery has been delivered to the actor instance subsystemB. The second line asserts that the value attribute objX.addr was given the same value as the first parameter of the result of the call. The assumption is that the results are always available when the postcondition is evaluated.

Finally, we have to show how an operation returning a result can be specified by an operation schema. In the postcondition that describes such an operation, the reply event is referred to by the keyword *result*, and from *result* one can navigate to the return parameters.

For example,

```
Operation: SubsystemB::makeAQuery (): Param1Type;
Post:
    result = ReplyToMakeAQuery ((param1 => Color::blue));
```

We could have equally replaced the last line with:

```
    result.param1 = Color::blue;
```

The result event is implicitly sent back to the sender (who made the call), e.g., the following is redundant and may be omitted:

```
    sender.sent (result);
```

5.7.2 Exceptions

Despite our assumption for reliable communications, there are often situations where the called actor cannot provide what was requested for. We will use exceptions for handling these situations. For instance, in the example of figure 9, if the scheduler can not return a request to be serviced it might throw an exception, rather than return some "dummy" value. We require that any actor requesting a service that can throw an exception must provide an exception handler. It may choose to pass it on, but this is to be asserted explicitly in a handler. We therefore propose to add an additional clause in the schema format called **Exceptions**. This clause is used to handle all exceptions stated in the **Sends** clause (associated with the **Throws** keyword).

The **Post** clause of the schema should be written in such a way that the functionality associated with exception handling is asserted within the **Exceptions** clause and not in the **Post** clause. In the case that the called operation throws an exception, instead of getting a result via the result rolename of the output event, the caller will receive an exception in its event queue, and the semantics of the Operation Schema's postcondition will be the conjunction of the **Post** clause and the **Exceptions** clause. It is possible to write a specification that conforms to this rule because in the **Post** clause, the expression, event.result->isEmpty (), is true if an exception occurred.

We demonstrate exception handling on a call to the scheduler. The handler deals with the case when the scheduler is unable to return a request to be serviced, and instead raises an exception called noRequests_e:

```
Post:
scheduler.sent (gnr) &
if gnr.result->notEmpty () then
    self.currentRequest = gnr.result.nextRequest
endif;
Exceptions:
noRequests_e () HandledBy
    self.mode = Mode::express;
```

The **Post** clause asserts that the scheduler actor, scheduler, is delivered gnr, an event of type GetNextRequest_r, and if there is a reply, then the system's current request is equivalent to the nextRequest parameter of the result. The **Exception** clause states that if the exception occurrence of type NoRequests_e is thrown as a consequence of a call made by the operation, then the condition after the **HandledBy** keyword is fulfilled.

6 Miscellaneous Issues

In this section, we discuss how packages of OCL constraints can be used in a similar way to libraries in programming languages and how we describe "creating" and "destroying" objects in Operation Schemas. And, we describe some issues related to the navigational style of OCL over n-ary associations.

6.1 Libraries and Packages

According to UML, packages can be used to store any kind of UML model elements. OCL constraints are a subtype of model element in the meta-model. It is therefore possible to define a package that contains only constraints. Furthermore, one could well imagine that a package could be used as a library or even just a common place to store related constraints. Clearly, packages could be useful to the OCL modeler for storing invariants, extensions to OCL types, parameterized predicates, functions, etc. For example, we could imagine "importing" a certain package of functions into a schema, like one would in Java, for example. Of course, the level of reuse would depend on how generic the constraints supplied by the package are.

6.2 Creation and Destruction of Objects

Instead of explicitly asserting that an object is created or destroyed with the execution of an operation, we rather define creation and destruction in terms of what is part of system state and what ceases to be part of system state. We judge an object as part of system state if and only if it has a composition link either directly or transitively with the system. For instance, an object that is a component of a component of the system is also part of the system state by transitivity. Thus, "creating" an object requires only that one asserts that a link was added to a composition association with the system (direct or transitive), and "destroying" an object requires that one asserts that a link was removed from a composition association with the system.

We have found that this approach simplifies the description of destruction in particular, because it abstracts away from a particular implementation interpretation, i.e., we could interpret removing the link between the object and the system as either instant destruction (a call to a destructor), or as flagging the garbage collector, or even as a prompt for the system to place the object back into the program's object pool.

6.3 Navigation

OCL was created with the main purpose of providing navigation of UML models and consequently it is asymmetric with respect to associations. OCL's style of navigation has quite some advantages, e.g. there are not too many operators and they are easy to understand, but there are also some drawbacks.

First of all, the addition of a new link between two objects can be easily misinterpreted. For example, in a postcondition an expression like the following:

```
cab.intRequests->includes (req)
```

means that there is a new link between cab and req in the HasIntRequest association. It can be easily misinterpreted as being a unidirectional link from cab to req, whereas the condition is strictly equivalent to:

```
req.requestingCabin->includes (cab)
```

More seriously, it is impossible to use the navigational notation for higher-order associations, and at least awkward to use it for handling attributes belonging to association classes.

7 Related Work

The idea of Operation Schema descriptions comes from the work on the Fusion method by Coleman et al. [6]. They took many ideas for Operation Schemas from formal notations, in particular, Z and VDM. The Operation Schema notation that we present here has a similar goal to the original proposal, but we have made notable changes to the style and format of the schema. Several proposals for formalizing Fusion models with Z and variants of Z have been proposed [2] [4]. One advantage of these approaches is that they can draw upon already existing analysis tools for Z.

Z [26] and VDM [15] are both rich formal notations but they suffer from the problem that they are very costly to introduce into software development environments, as is the case with most formal methods, because of their high requirements for mathematical maturity on the user.

The Z notation is based on set theory and classical first-order predicate logic. An interesting feature of the Z specification language is the schema notation. A schema can be viewed as an encapsulated structure, associated with some properties. Using the schema notation, it is possible to specify parts of a system separately, and then compose the specifications for the parts to obtain the specification of the whole system. Schemas are commonly used in Z to represent types, state spaces and operations.

In contrast to Z, which is strictly a specification notation, VDM offers notations that provide a wider treatment of the software lifecycle. VDM supports the modeling and analysis of software systems at different levels of abstraction. Both data and algorithmic abstractions expressed at one level can be refined to a lower level to derive a concrete model that is closer to the final implementation of the system. A VDM specification written in assertional style can be refined into another VDM specification written using statements, i.e., VDM has imperative programming constructs as part of its notation. An even more complete treatment of the software development lifecycle is offered by the B-method [1], which uses a formalism that has similarities to both Z and VDM. It uses a unified notation for specification, design and implementation and is supported by the B-toolkit which provides tool support for specification, animation, design, proof obligation generation, automatic and interactive proof, and code generation.

The Catalysis approach [8], developed by D'Souza and Wills, provides action specifications. Catalysis defines two types of actions: localized and joint actions. Localized actions are what we would term operations in our approach and joint actions are related to use cases. In the endeavor to support controlled refinement by decomposition through a single mechanism, Catalysis defines actions, which can be decomposed into subordinate actions, at a lower-level of abstraction, or composed to form a superordinate action, at a higher-level of abstraction. Furthermore, Catalysis defines joint actions to describe multi-party collaborations, and localized actions to describe strictly the services provided by a type. However, joint actions lack the ability of goal-based use cases to describe stakeholder concerns due to the focus of pre- and postconditions on state changes and not the goals of the participants/stakeholders. The activity of assuring stakeholder concerns when writing use cases is often a source for discovering new requirements and business rules. It is for these reasons that we did not merge use case descriptions and pre- and postcondition descriptions of operations, but instead chose to keep them separate.

Meyer [19] proposes design-by-contract; it is an assertion language that is integrated into the Eiffel object-oriented programming language. Pre- and postconditions are placed in class methods: *require* assertions (preconditions) are checked before their respective method is executed and *ensure* assertions (postconditions) are checked after the execution of the method; if either assertion fails then an exception is raised. All assertions are made on the program state and for each class. Therefore assertions are numerous and limited to the abstraction level of the implemented program. Design-by-contract is also complemented by the BON method [28]. The BON assertion language has similarities to OCL [20].

8 Conclusion

The goal of this paper was to motivate and justify a number of enhancements and interpretations that we made to UML's Object Constraint Language while developing an approach for pre- and postcondition assertions. We proposed a number of modifications for making OCL more effective when used by developers for writing and reading pre- and postconditions: incremental descriptions, aggregates, structuring techniques, etc. Also, the paper discussed our proposal for specifying events and exceptions and delivering them to actors.

We defined a list of criteria, detailed in section 1, to measure our approach and guide its design. Currently, we believe our approach fulfils, more or less, the first five criteria and goes someway in fulfilling the last three ones—part of our future work. Our approach conforms to UML (criterion 1) and we propose a model that is sufficiently precise and consistent that the transition to design can be performed in a systematic manner (criterion 2). Also, we believe that we were able to show that with our approach usability does not necessarily have to be traded-in against rigor. For example, we believe that the application of the minimum set principle and our frame assumption makes it easier to formulate correct postconditions (criterion 3). We proposed parameterized predicates and functions to manage reuse and to support modularity (criterion 4). Our model offers both operational and representational abstraction (criterion 5): the pre- and postconditions avoid design details thanks to their declarative description and because the state of the system is defined in terms of domain concepts and not software components. Currently, we have a prototype tool that supports our approach by checking both syntax and type correctness (criterion 6). Our ideas for modeling performance constraints on the SIP (criterion 7) and concurrent operations (criterion 8) are published elsewhere [25].

Our approach has been successfully taught to students and practitioners and used in a number of small-to-medium sized projects. This leads us to believe that Operation Schemas based on OCL are not only a powerful, but indeed a usable mechanism for precisely describing operations.

For examples of our approach applied to several case studies see [35].

References

[1] J. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] K. Achatz and W. Schulte. *A Formal OO Method Inspired by Fusion and Object-Z*. In J. P. Bowen, M. G. Hinchey, and D. Till (eds.): ZUM'97: The Z Formal Specification Notation, LNCS 1212 Springer, 1997.

[3] A. Borigda, J. Mylopoulos and R. Reiter. *On the Frame Problem in Procedure Specifications*. IEEE Transactions on Software Engineering, Vol. 21, No. 10: October 1995, pp. 785-798.

[4] J-M. Bruel and R. France. *Transforming UML models to formal specifications*. Proceedings of the OOPSLA'98 Workshop on Formalizing UML: Why? How?, Vancouver, Canada, 1998.

[5] P. Chen; *The Entity-Relationship Model—Toward A Unified View of Data*. ACM Transactions on Database Systems, 1(1), 1976, pp. 9-36.

[6] D. Coleman et al. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[7] S. Cook, A. Kleppe, R. Mitchell, J. Warmer, and A. Wills. *Defining the Context of OCL Expressions*. Second International Conference on the Unified Modeling Language: UML'99, Fort Collins, USA, 1999.

[8] D. D'Souza and A.Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley 1998.

[9] J. Daniels et al. *Panel: Cracking the Software Paradox*. OOPSLA 2000 Companion from the Conference on Object-Oriented Programming, Systems, Languages, and Application, USA, 2000.

[10] K. Finney, N. Fenton, and A. Fedorec. *Effects of Structure on the Comprehensibility of Formal Specifications*. IEEE Proc.-Softw. Vol. 146, No. 4, August 1999.

[11] D. Firesmith. *Use Case Modeling Guidelines*. Proc. 30th Conference on Technology for Object-Oriented Programming Languages and Systems (TOOLS-30), pp. 184-193, IEEE Computer Society, 1999.

[12] M. Fowler; *Use and Abuse Cases*. Distributed Computing Magazine, 1999 (electronically available at http://www.martinfowler.com/articles.html).

[13] M. Glinz; *Problems and Deficiencies of UML as a Requirements Specification Language*. Proceedings of the Tenth International Workshop on Software Specification and Design, San Diego, 2000, pp. 11-22.
[14] J. Guttag et al. *The Larch Family of Specification Languages*. IEEE Trans Soft Eng 2(5), September 1985.
[15] C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.
[16] M. Kandé and A. Strohmeier. *Towards a UML Profile for Software Architecture Descriptions*. UML 2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, S. Kent, A. Evans and B. Selic (Eds.), LNCS (Lecture Notes in Computer Science), no. 1939, 2000, pp. 513-527.
[17] B. Kovitz; *Practical Software Requirements: A Manual of Content and Style*. Manning 1999.
[18] C. Morgan. *Programming from Specifications*. Second Edition, Prentice Hall 1994.
[19] B. Meyer. *Object-Oriented Software Construction*. Second Edition, Prentice Hall, 1997.
[20] R. Paige and J. Ostroff. *A Comparison of the Business Object Notation and the Unified Modeling Language*. UML '99 - The Unified Modeling Language: Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Robert France and Bernard Rumpe (Eds.), LNCS (Lecture Notes in Computer Science), no. 1723, 1999, pp. 67-82.
[21] B. Potter, J. Sinclair and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.
[22] M. Richters and M. Gogolla. *On Formalizing the UML Object Constraint Language OCL*. In Tok Wang Ling, Sudha Ram, and Mong Li Lee, editors, Proc. 17th Int. Conf. Conceptual Modeling (ER'98), pages 449-464. Springer, Berlin, LNCS Vol. 1507, 1998.
[23] S. Sendall and A. Strohmeier. *UML-based Fusion Analysis*. UML '99 - The Unified Modeling Language: Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Robert France and Bernard Rumpe (Ed.), LNCS (Lecture Notes in Computer Science), no. 1723, 1999, pp. 278-291, extended version also available as Technical Report (EPFL-DI No 99/319).
[24] S. Sendall and A. Strohmeier. *From Use Cases to System Operation Specifications*. UML 2000 — The Unified Modeling Language: Advancing the Standard, Third International Conference, S. Kent and A. Evans (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, pp. 1-15; Also available as Technical Report (EPFL-DI No 00/333).
[25] S. Sendall and A. Strohmeier. *Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML*. <<UML>> 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, Fourth International Conference, Toronto, Canada, October 1-5, Martin Gogolla (Ed.), Lecture Notes in Computer Science, Springer-Verlag, to be published in 2001. Also available as Technical Report EPFL-DI No 01/367.
[26] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.
[27] A. Strohmeier and S. Sendall. *Operation Schemas and OCL*. Technical Report (EPFL-DI No 01/358), Swiss Federal Institute of Technology in Lausanne, Software Engineering Lab., 2001.
[28] K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*. Prentice-Hall, 1995.
[29] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley 1998.
[30] J. Wing. *A Two-tiered Approach to Specifying Programs*. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

Electronic Resources

[31] Klasse Objecten. *OCL Center: OCL Tools*. http://www.klasse.nl/ocl/index.htm
[32] OMG Unified Modeling Language Revision Task Force. *OMG Unified Modeling Language Specification*. Version 1.3, June 1999. http://www.celigent.com/omg/umlrtf/
[33] Software Engineering Lab., Swiss Federal Institute of Technology in Lausanne. *The Fondue Method*. http://lglwww.epfl.ch/research/fondue/
[34] Software Engineering Lab., Swiss Federal Institute of Technology in Lausanne. *Operation Schemas*. http://lglwww.epfl.ch/research/operation-schemas/
[35] S. Sendall. *Specification Case Studies*. http://lglwww.epfl.ch/~sendall/case-studies/