Specifying System Behavior in UML

Shane Sendall and Alfred Strohmeier

Swiss Federal Institute of Technology in Lausanne
Department of Computer Science
Software Engineering Laboratory
1015 Lausanne EPFL, Switzerland

email: {Shane.Sendall, Alfred.Strohmeier}@epfl.ch
fax: ++ 41 21 693 5079

ABSTRACT The purpose of the paper is to present our approach for specifying system behavior. Our approach is based on operation schemas and a system interface protocol (SIP). Operation schemas describe the functionality of system operations by pre- and postconditions; they are written in the Object Constraint Language (OCL), as defined by the Unified Modeling Language (UML). A SIP describes the temporal ordering of the system operations by a UML protocol statemachine. Operation schemas are hybrids of formal specifications approaches such as Z and VDM : hybrid in the sense that they are based on their formal counterparts but are targeted to developers that are more comfortable with procedural programming languages rather than declarative languages.

Our approach offers a middle ground between the informal descriptions of use cases and the solution-oriented models of object interaction in UML. We believe that declarative behavioral specification techniques, like the one proposed in this paper, lead to more confidence in the quality of the software because they allow one to reason about system properties.

KEYWORDS Unified Modeling Language (UML), Object Constraint Language (OCL), Precondition, Postcondition, Formal Specification, Object-Oriented Software Development.

1 Introduction

Our increasing reliance on software intensive systems in everyday life is forcing us to re-evaluate the importance of producing software that functions correctly. The development of this software infrastructure will require us to take more care in assuring that high-quality software is produced and put in place [20]. We believe that raising the quality of software can be facilitated by better descriptions of system behavior. We identified the following criteria for evaluating behavioral descriptions for their effectiveness within a context of main-stream software development:

- The descriptions should be compatible with industry practices and standards.
- The descriptions should be targeted towards the ease of use by the developer, i.e. it should be simple, concise, understandable, modular, malleable, etc.
- The description should be precise so that it can be used as a clear and unambiguous contract for later activities.
- The time needed to develop the description should not compromise the cycle time of the development process, i.e., the description should not be heavy in comparison to the other models of development.
- The description should be scalable to manage large problems, and it should be possible to focus just on the essential problem without getting caught up on less essential details, thus allowing one to manage complexity and size.

- The description should be conducive to verification and validation of the end-product. Clearly, the level of analysis should be relative to how critical the correct functioning of the software is within its environment.
- The description should clarify "quantifiable" non-functional requirements of the system in an integrated and traceable way, such as performance constraints.
- The description should be capable of capturing inherent concurrent properties of the system and quality of service properties as found when modeling continuous streams in multimedia systems.

At the end of 1997, the Unified Modeling Language (UML) was standardized by the Object Management Group (OMG) [19]. UML is an informally founded language that offers a rich set of notations for modeling both the static and dynamic aspects of an object-oriented system under development. Currently in industry much of what would be loosely classified as system specification is performed with use cases. Use cases are an excellent tool for capturing behavioral requirements of software systems. They are informal descriptions, almost always written in natural language, and consequently they lack rigor and a basis to reason about system properties.

On the other hand, formal specification approaches such as Z [16] and VDM [8] propose declarative specifications of system behavior by pre- and postconditions. They provide the capability to reason about system properties, and they promote rigor and precision. They define the system behavior by stating changes of the system on a conceptual model. Use cases, alternatively, define the interactions between the system and external actors, in terms of actor goals, stakeholder concerns and system responsibilities. Formal specifications also normally require a high-level of mathematical maturity to read and understand, and therefore are not primarily targeted towards stakeholder comprehension, as is the case for use cases.

Our approach uses operation schemas and a system interface protocol (SIP), which are complementary to use cases [15]. Operation schemas describe the functionality of system[1] operations by pre- and postconditions; they are written in the Object Constraint Language (OCL), as defined by the Unified Modeling Language (UML) specification. An SIP describes the temporal ordering of the system operations with a UML protocol statemachine. Our approach can be used as a middle ground in UML between the informal descriptions of use cases and the solution-oriented models of object interaction.

Operation schemas are hybrids of formal specifications approaches such as Z and VDM: hybrid in the sense that they are based on their formal counterparts but are targeted to developers that do not necessarily have a strong background in mathematics and are more comfortable with procedural programming languages rather than declarative languages. Operation schemas allow one to precisely describe the services provided by the system and due to their declarative nature are less likely to embody premature design decisions. They allow one to formalize business rules and reason about system properties.

An SIP is a UML protocol statemachine that focuses on the temporal ordering of the system operations only, and therefore the usage of the UML state diagram notation is very specific, and only a limited use is made of the notation. Whereas operation schemas describe the services offered by the system, the SIP describes the allowable sequencing of these services. The two are refined from use cases and they combine to define a precise specification of system behavior. An approach for mapping use cases to operation schemas has been proposed in [15]. To see how this work fits into a software development "analysis" activity, the reader is referred to [14].

1. A system in this sense could as well be a subsystem in a larger system.

This paper puts particular focus on defining the operation schema, covering its syntax, informal semantics, and basic guidelines for use, and shortly describes the SIP. We illustrate the use of our approach with an example of an elevator control system.

The paper is composed in the following way: section 2 describes the elevator control system example that is used in varies places throughout the paper; section 3 provides an introduction to operation schemas and OCL and shows an operation schema for the elevator control system; section 4 goes further into the style and semantics of operation schemas; section 5 describes the system interface protocol; section 6 provides a discussion on current and future work; section 7 discusses related work; and finally section 8 concludes the paper.

2 Elevator Control Example

For illustrating our approach, we will describe an elevator control system. The system controls multiple lift cabins that all service the same floors of a building. There is only one button on each floor to request a lift cabin. 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 an cabin to go up, go down or stop. In this example, we assume that an elevator cabin's braking distance is negligible. The system may ask an elevator 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 of how the user goes from one floor to another with a lift could be: A user calls the lift. An available lift comes to the floor of the requesting user to pick him/her up. The lift stops and opens its door. The user gets in and requests a destination floor. The lift closes its door and goes to the destination corresponding to the request made by the user. The lift stops and opens it door. The user leaves the lift at the destination floor.

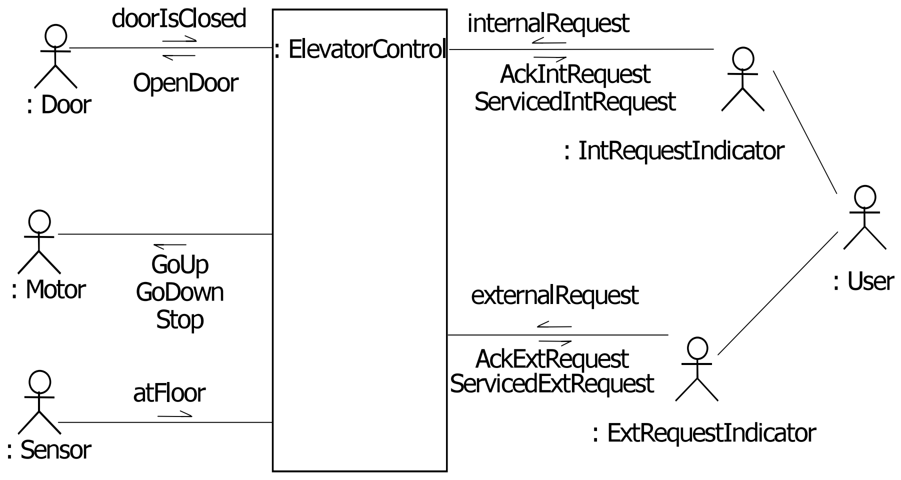

Fig. 1. Elevator Control System Context Model

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 [15]. 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 1. The System Context Model 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 model 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 communication paths often clarifies the consistent overall working of a system.

The analysis-level class model for the elevator control system is shown in figure 2. It shows all the domain concepts and relationships between them. Inside the system there are five domain 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 collection of internal requests to a particular cabin, HasCurrentRequest links a cabin to its current request, hasExtRequest models the collection of all external requests issued by users, 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 form inside the system, will be the only use of <<id>> stereotyped associations.

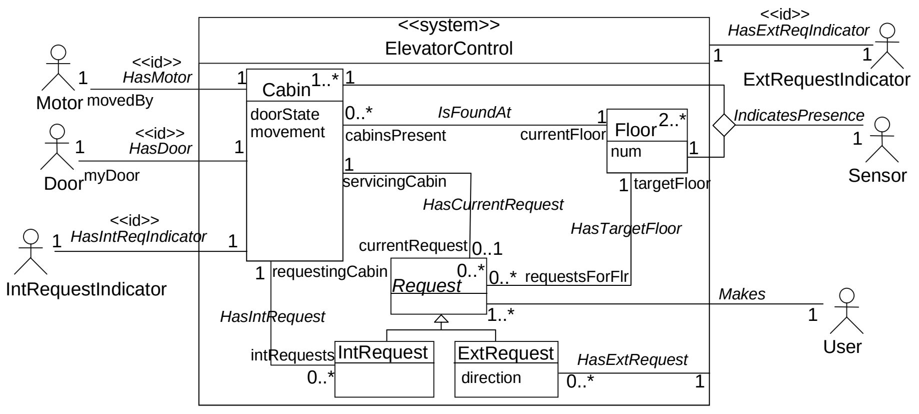

Fig. 2. Elevator Control System Class Model

3 Operation Schemas and OCL

An operation schema describes the effect of the operation on an abstract state representation of the system and by events sent to the outside world. It is written in a declarative form that abstracts from the object interactions inside the system which will eventually realize the operation. It describes the *assumed* initial state by a precondition, and the change in system state after the execution of the operation by a postcondition. Operation schemas use UML's OCL formalism, which was built with the purpose of being writable and readable by developers. Operation schemas as we define them here specify operations that are assumed to be executed atomically.

The system model is reactive in nature and all communication with the environment is achieved by asynchronous input/output events, termed signals in UML[1]. All system operations are triggered by input events, normally 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 are themselves described in the system class mode (fig. 2). 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 required by an operation.

The class model is used to describe all the concepts and relationships in the system, and all actors that are present in the environment. Therefore, the class model as we define it here is not a design class model. Classes and associations model concepts of the problem domain, not software components. Objects and association links hold the system state. Classes do not have behavior; the decision to allocate operations or methods to classes is deferred until design.

The standard template for an operation schema is shown in figure 3. 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. 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.

Operation: This clause displays the system name followed by the operation name and parameter list.
Description: This clause provides a concise description of the operation written in natural language.
Notes: This clause provides additonal comments.
Use Cases: This clause contains cross-references to superordinate use case(s).
Scope: This clause declares the classes and associations of the class model that are used in the schema.
Declares: This clause provides declarations of all constants and variables designating objects, datatypes, object collections, and datatype collections used in the **Pre** and **Post** clauses.
Sends: This clause specifies which kinds of events are sent to which actor types. It is also possible to declare event instances and event collections.
Event Order: This clause defines constraints on the order of events output by the operation.
Pre: This clause is the operation's precondition, written in OCL. It contains a boolean expression. The precondition cannot refer to parameters of the operation.
Post: This clause is the operation's postcondition, written in OCL. It contains a boolean expression. If the precondition is true, then the operation terminates and the postcondition is true after the execution of the operation; if the precondition is false, the behavior of the operation is not defined by the schema. This is also the only clause that uses the notation @pre for referring to the state preceding the operation invocation.

Fig. 3. Operation Schema Format

3.1 Presentation of OCL

UML [19] defines a navigation language called the Object Constraint Language (OCL) [17], a semi-formal language for writing expressions whose principles are based on set theory. 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.

1. According to UML, use cases use signals for the communication between the system and actors.

OCL is a declarative language. An OCL expression has no side effects, i.e. an OCL expression constrains the system by observation rather than simulation of the system. When describing operations, an OCL expression is evaluated on a consistent system state, i.e. no system changes are possible while the expression is evaluated. 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.

We now highlight the atFloor operation schema, shown in figure 4. 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; if so, it will drop off and/or pick up the user(s), otherwise the system will let the lift continue to its destination.

The **Declares** clause defines a local boolean variable, makeStop, which results in true if there is an internal request or external request (that is requesting the same direction as the lift is currently going) for the supplied floor f. The **Sends** clause shows that instances of the event types Stop, GoUp, GoDown, OpenDoor, ServicedExtRequest, ServicedIntRequest may be sent to the indicated actors and that Stop and OpenDoor have named instances. The **Event Order** clause 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. The **Pre** clause states that the cabin cab has a currentRequest, i.e., cab is currently servicing a request, and cab is moving.

The dot notation usually results in a set of objects or values, including the special cases of a single element or an empty set. For instance, self.cabin is the set of all cabins in the system, self denoting the system instance. 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.extRequest denotes the set of external requests that can be reached by navigating from self (the system instance) on the hasExtRequest association.

The arrow operator is used only on collections, in postfix style. The operator following the arrow is applied to the previous “term”. For instance, dropOffRequest ->union (pickUpRequest) results in a set consisting of the union of the two sets dropOffRequest and pickUpRequest.

The first line of the **Post** clause states that the cabin is now found at floor f. 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. Note that the expression, self.request->excludesAll (reqsForThisFloor), not only removes the serviced request objects from the set of Request instances, but deletes also all the association links targeting one of these objects from the associations IntRequest, ExtRequest and CurrentRequest. An explanation of our frame assumption for operation schemas which explains this sort of implicit removal is provided in section 4. Also, the & operator used throughout the schema is a shorthand for logical “and”. In the Post clause, sending events is described by stating that an event instance was delivered to the appropriate actor instance. For example, the third line of the postcondition states that the actor instance cab.movedBy, denoting a navigation from the cabin to its motor via the HasMotor association, has had an event instance called stop placed in its events queue. Looking further at the OCL notation, an expression, such as cab.doorState = #open, means that the attribute, doorState, of the object cab has the value open (the ‘#’ indicates an enumerated type value) after the execution of the operation.

```
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 Case(s): TakeLift;
Scope: Cabin; Floor; Request; IntRequest; ExtRequest; HasIntRequest; HasExtRequest; Has-
CurrentRequest; HasTargetFloor; IsFoundAt;
Declares:
  directionHeading: Direction ::= if self.externalRequest->includes (cab.currentRequest) then
                                  cab.currentRequest.direction else cab.movement endif;
  dropOffRequest: Set (IntRequest) ::= cab.intRequests->select (r | r.targetFloor = f);
  pickUpRequest: Set (ExtRequest) ::= self.extRequest->select (r | r.targetFloor = f &
                                        r.direction = directionHeading);
  reqsForThisFloor: Set (Request) ::= dropOffRequest->union (pickUpRequest);
  makeStop: Boolean ::= reqsForThisFloor->notEmpty;
Sends:
  Motor::{Stop, GoUp, GoDown}, Door::{OpenDoor},
  ExtRequestIndicator::{ServicedExtRequest}, IntRequestIndicator::{ServicedIntRequest};
  stop: Stop, open: OpenDoor;
Event Order:
  Sequence {stop, open}; -- the output events are sent in the order stop followed by open
Pre:
  cab.currentRequest->notEmpty & -- cab was going somewhere
  cab.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).events->includes (stop) & -- stop sent to cab motor
      cab.movement = #stopped &
      (cab.myDoor).events->includes (open) & -- open sent to door
      cab.doorState = #open &
      self.request->excludesAll (reqsForThisFloor) & -- removed request(s) for this floor
      if pickUpRequest->notEmpty then
          (self.extReqIndicator).events->includes (ServicedExtRequest' (
              callingFlr => pickUpRequest.targetFloor, dir => pickRequest.direction))
      endif &
      if dropOffRequest->notEmpty then
          (self.intReqIndicator).events->includes (ServicedIntRequest' (
              destFlr => dropOffRequest.targetFloor))
      endif
  endif
```

Fig. 4. atFloor Operation Schema for Elevator Control System

4 Style and Semantics of Operation Schemas

The state of the system is defined by the system class model. It is defined by the states of all instances of all classes and all association links. In addition to the production of output events, an operation schema describes, by pre- and postconditions, how the system state is changed by the execution of the operation. An implementation of the operation must, therefore, obey the postcondition; otherwise the implementation has incorrect behavior. However, if the implementation of the operation is doing more than what is strictly needed to satisfy the postcondition, e.g. creating some additional objects, or changing the states of some objects which are outside the frame of the operation, then it would also satisfy the postcondition. We will discuss this issue in subsection 4.1.

Operation schemas have a procedural programming-like style, which is above and beyond the operational style of OCL. The following list displays the additions to OCL that lead to a procedural style; some are notational shorthands, others are more semantics enhancements:

- Frame assumption – We use a pragmatic frame assumption, discussed in section 4.1.
- Branching – Case distinction is realized by if-then-else conditions rather than implies conditions. The usefulness of this choice becomes evident when schemas have a large number of case distinctions. Also, our pragmatic frame assumption allows the use of if-then conditions, and we also allow if-then-elsif conditions.
- Aggregate notation – We allow the use of an Ada-style aggregate notation for denoting composite values. The set of components of a record, the value attributes of an object, and the parameters of an event all correspond to composite values. An aggregate can be written using named associations, i.e. a value is associated with each component denoted by its name, e.g. the attribute values of a cabin object:
  ```
  (doorState => #closed, movement => #stopped)
  ```
- Incremental operators – We apply the principle of minimal sets to postconditions and also propose a similar principle for “incremental” numerical operations. These principles and notations are discussed in section 4.2.
- Structuring mechanism – To better support readable schemas and to provide encapsulation of commonly recurring conditions, we use parameterized predicates. Parameterized predicates can be seen as a parameterized piece of the postcondition. Consequently, parameterized predicates can use the suffix ‘@pre’.

4.1 Frame Problem

The frame of the specification is the list of all variables that can be changed by the operation [11]. 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 the same. These approaches soon become cumbersome to write and error-prone, particularly for specifications that have complex case distinctions. One approach that avoids this extra work is to imply a “... and nothing else changes” rule when dealing with specifications [2]. 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. However, this assumption does not work in the cases of implicit removal and implicit override. Implicit removal is applied to preserve the consistency of the class model. E.g. when an object is removed from the system state, all association links it participates in must also be removed. Without the “implicit removal” hypothesis, it would be necessary to explicitly state all these system changes. For an example, let’s consider the following, very simple postcondition:

```
self.cabin = self.cabin@pre->excluding (cab)
```

If we strictly apply the frame assumption "... and nothing else changes", the result would be an inconsistent system state whenever cab has association links to requests, floors etc.; the respective associations must therefore be changed.

Taking this into account, we modify our frame assumption to "... and nothing else changes, except in the case of an implied removal or implied override". In these two cases we need to loosen the assumption to allow the implicit update of the corresponding associations.

4.2 Principle of Minimal Set

We propose to define the semantics of an operation schema by applying the idea of the minimal set condition. For each class and each association, we will consider their sets of instances and links, and claim that these are all minimal sets after execution of the operation. Otherwise stated, if C is a class, if Set(C)@pre is its set of its instances before the execution of the operation, and Set(C) is its set of its instances after the execution of the operation, then Set(C) is the minimal set containing Set(C)@pre and fulfilling the postcondition. Intuitively, Set(C) can be constructed by adding to Set(C)@pre all instances of C created by the operation. The same kind of idea can be applied to the links of an association A; Set(A) is then the minimal set containing Set(A)@pre and fulfilling the postcondition. The rule must hold for all classes and associations. As explained in the previous section, there is a slight problem when we allow for the destruction of objects or removal of association links. For defining the semantics of the operation schema, the idea is then to gather the deleted entities into a temporary set, and rephrase the rule in the following way: let C be a class, let's denote by Minus(C) the set of instances of C destroyed by the operation, then Set(C) $\cap$ Minus(C) is empty, and Set(C) $\cup$ Minus(C) is the minimal set containing Set(C)@pre.

For example, if we state in the postcondition the condition,

```
self.extRequest->includes (f)
```

it is equivalent to:

```
self.extRequest = self.extRequest@pre->including (f)
```

at least as long as no other statements have been made about the state of self.extRequest.

Minimal sets can be very useful for stating postconditions incrementally. For example, we could define a fragment of an imagined operation called swapFloors, which exchanges one floor with another in the association ExtRequest:

```
self.extRequest->includes (f) &
self.extRequest->excludes (g)
```

Clearly, this would be much harder to state if we had to write a single equality. Minimal sets, in this context, have a similar effect to re-dashing of schemas when composing them in Z [13]. The minimal set principle has been used in the atFloor operation schema shown in figure 4.

We can use an idea similar to minimal sets for object attributes that are of a number type. We propose to use the operators, "+=" and "-=". Thus, the value of the object attribute 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 postcondition that refer to the object attribute, and minus all the right-hand sides of all "-=" operators that refer to the object attribute. For example:

```
x += 5 &
x -= 4
```

is equivalent to:

```
x = x@pre + 1
```

Such a notation is especially useful when there are many case distinctions.

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

5 System Interface Protocol

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 5. It consists of two parallel substates. The top-most substate models the activity of processing external requests. The dashed line shows that it works in parallel with the lift activities.

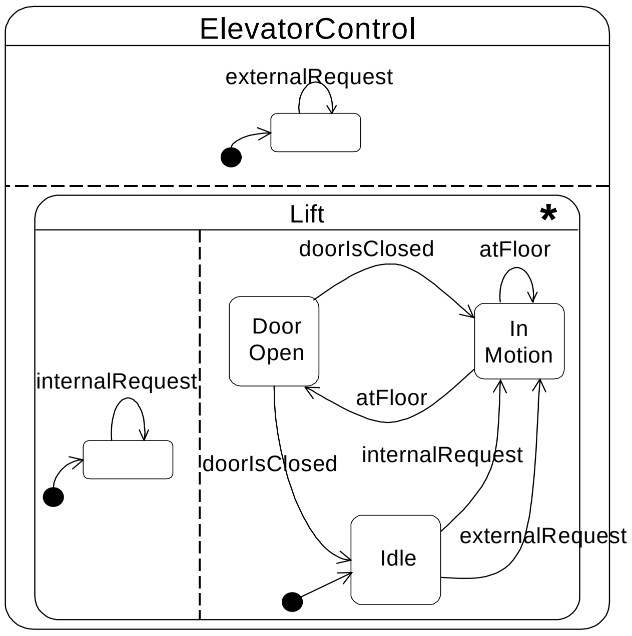

Fig. 5. Elevator Control System Interface Protocol

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 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.

The SIP complements the operation schema descriptions because it describes the temporal ordering of the system operations. The SIP offers two complementary features in addition to operation schemas. Firstly, it acts as a guard to the operation, i.e., the SIP must be in a correct state for the operation to have permission to execute. Secondly, the SIP allows one to state all possible operation sequences, which allows one to test for valid and invalid operation sequences.

Also, the SIP can be useful for describing performance constraints. For example, we could attach a real-time constraint on the Lift submachine which states that a lift is not allowed to have its door open for more than 5 seconds. This could be specified by placing a constraint on the Door Open state, which places a 5 second deadline on a doorIsClosed event after an atFloor event enters the Door Open state. Further investigation of the SIP's ability to model performance constraints is current work. We also hope to take into account such performance constraints when architecting the system solution [10].

6 Discussion of Current and Future Work

The ultimate aim of this work is a modeling technique capable of specifying distributed systems and their components. In this section, therefore, we discuss our initial ideas on how to use operation schemas for modeling mutually concurrent operations and blocking calls to other actors or subsystems.

6.1 Modeling Blocking Calls

Figure 6 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. The second approach (bottom) achieves the same result by asynchronous events. Consequently two operation calls are made to subsystemA, as opposed to a single one in the first approach.

The question is what granularity of description is more natural. We believe that both possibilities are important, although we stress that care should be taken not to have too coarse grained schemas. Clearly, a pre- and postcondition description of a single operation for the whole system would not be very enlightening.

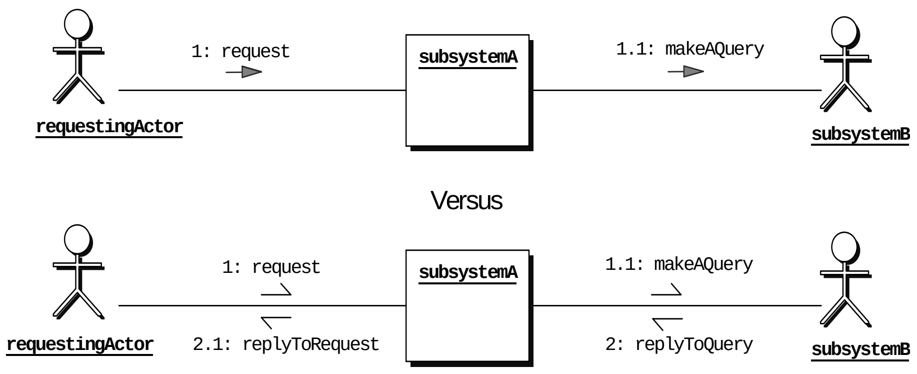

Fig. 6. Blocking Call versus Non-Blocking Events

In situations like the one shown in figure 6, blocking calls are more natural and therefore we propose to introduce them into operation schemas. We propose to model blocking calls in a similar way to asynchronous events: the calls are sent to actors as events, but in addition the output event is associated to a result event—the reply. This proposal assumes reliable communication between the system and the actors, and also that the actor will respond, i.e., is not down.

We propose to specify a blocking call by stating in the postcondition that the event was sent to an actor and that the system received a reply event. Also, the reply event must be related to the original event by a predefined association that can be traversed with the role name result. And therefore the results returned from the call can be accessed directly from the reply event.

For example, a piece of a postcondition that states that a blocking call occurred during execution could be:

```
actorA.events->includes (blockingEvent) &
objX.addr = blockingEvent.result.param1 -- the event result has possibly many return parameters
```

The first line states that the event blockingEvent has been sent to actorA. The second line states that the datatype attribute addr of objX has the value of the first parameter of the result. Implicitly, all blocking calls have an associated result, which is itself an event that can have any number of parameters (modeled as attributes).

6.2 Modeling Concurrent Operations by Schemas

We propose to release the interference-free constraint on operations and to allow the specification of operations that execute concurrently[1]—operations that are possibly changing the state of the system in parallel. The rely/guarantee conditions of Cliff Jones [9] allow one to state under what conditions the postcondition makes sense in the presence of concurrency. If an operation is invoked in a situation when the precondition is false, or if during the execution of the operation the rely condition becomes false, then the specification does not state what the outcome should be. Otherwise the postcondition will be true at the end of the execution and the guarantee condition will have been maintained throughout. More specifically, the conditions have the following meaning in a specification:

Rely: All changes to system state performed concurrently by other operations must satisfy the rely condition (i.e., it is a requirement on the outside world).

Guarantee: All changes to system state performed by this operation satisfy the guarantee condition (i.e., an invariant that holds during the execution of the operation).

To deal with concurrent access to the system state, we propose an additional clause in the schema that is used to state which variables require mutual exclusion. This clause is called **Mutex**. We also propose to only state mutual exclusion constraints on the variables that are changed by the operation. We therefore postpone the decision which synchronization policy to use. For example, one such policy could be many readers/one writer with writer priority. Taking this simple view on operation synchronization allows one to postpone the issues of fairness and liveliness to later stages of development.

When writing operation schemas for concurrent operations, the easiest way to formulate the postcondition is to write it like we would do it were no interference by other concurrent operations. Unfortunately, this is not possible when describing changes to a shared variable. Indeed, such a variable can be changed by a concurrent operation, and the meaning of @pre is therefore undefined, and should be avoided. In the endeavor to make as few changes as possible to the interference-free style of writing schemas, we propose to use functions that are not bound to the state at time @pre. For example, instead of x = x@pre + 5, we propose to use x += 5, with the meaning that x has a value of 5 greater than it was before the addition. Similarly, we overload the definition of the other incremental operators.

1. Clearly, this will require us think about how this affects our notion of blocking calls.

Taking into account the previous proposals, we will show as an example the transfer operation of a banking system, shown in figure 7. The transfer operation schema specifies that the system takes a specified amount from a source account and places it on a specified destination account. To allow concurrent execution, we specify the rely and guarantee conditions and state mutual exclusion constraints: both src.balance and dest.balance must be updated in mutual exclusion.

```
Operation: Bank::transfer (src:Account, dest:Account, amount: Money)
Description: The system takes amount from the source account and places it on the destina-
tion account, iff the source account has sufficient funds.
Notes: This operation may execute concurrently with other operations that also work on
accounts.
Mutex: -- proposed additional clause
  mutex{src.balance, dest.balance};
Pre:
  src.balance >= amount
Rely: -- proposed additional clause
  src.balance >= amount &
  self.state = #deposit_withdraw_possible
Guarantee: -- proposed additional clause
  src.balance >= 0
Post:    -- we specifically propose a post that is very similar to the one that
         -- would have been written for a sequential system
  src.balance -= amount &
  dest.balance += amount
```

Fig. 7. A possible specification of a transfer for a banking system

The **Rely** clause states that during the whole execution of this operation the balance of the source account never goes below amount and that the state of the bank is continually in a state that allows deposits and withdrawals. The **Guarantee** clause states that the balance of the source account will not go below zero if the precondition and the rely condition hold. The **Post** clause states that the balance of the source account has the amount less on it than it had before the subtraction was executed; it also states that the balance of the destination account has the amount in addition to what it had before the addition was executed; and finally it states that an output event was sent to a certain actor (inferred by the comment, not included for reasons of conciseness).

As seen on this example, an operation is just a series of accesses and atomic update operations at a finer level of granularity. This leads us to the fact that the enforcement of the rely condition during the whole execution of the operation may be a constraint too strong in certain cases. For example, one could imagine a scenario where during the transfer operation the balance of the source account may in fact go below amount (and therefore falsify the condition), but just before the debit is executed another operation puts the source account's balance above amount. Clearly this scenario is valid behavior that is not allowed by the schema. This is a consequence of stating concurrency constraints at a coarse grain, in a top-down approach.

7 Related Work

The idea of operation schema descriptions comes from the work on the Fusion method by Coleman et al. [5]. 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 [1] [3].

The advantage of these approaches is that they can draw upon already existing analysis tools for Z.

An approach to unite the use case descriptions with formal specifications, allowing first the capture of the functional requirements by use cases and then making them more precise by formal specification and possible refinement afterwards, has been proposed by Petre et al. [12].

Z [16] and VDM [8] 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. On the other hand, OCL, the language used in operation schemas, has the advantage of being a relatively small and mathematically less-demanding language that is targeted at developers. One of the secrets of OCL's simplicity is that it uses navigation and operators manipulating collections rather than relations. Also, OCL was created for the distinct and sole purpose of navigating UML models, making it ideal for describing constraints and expressing predicates when a system is modeled with the UML.

The Catalysis approach [7], developed by D'Souza and Wills, provides action specifications which, of all related work, is the closest to ours. 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 and obligations of the participants/stakeholders. The activity of assuring stakeholder concerns, when writing use cases, is often a source for discovering new business rules. It was for these reasons that we preferred to separate use case descriptions from pre- and postcondition descriptions of operations.

Tool support for OCL is becoming more prevalent. A full list of tool support for OCL can be found at [18].

8 Conclusion

The goal of this paper was to show how the combination of operation schemas and an SIP can be used to provide a precise specification of system behavior by using the UML. When designing the style of operation schemas, we defined a list of criteria, section 1, to measure such an approach. Currently, our approach does not fulfil all the criteria mentioned, in particular the last three criteria are the focus of current/future work. However, we hope that we were able to show that the semantics of operation schemas are well defined, and that usability does not necessarily have to be traded-in against rigor. For example, we believe that the application of the minimal set principle and our frame assumption makes easier the formulation of correct postconditions.

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, in particular, are not only a powerful, but indeed a usable mechanism for describing precisely system-level operations.

9 References

[1] 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.

[2] 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.

[3] 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.

[4] A. Cockburn. *Writing Effective Use Cases*. The Crystal Series for Software Engineers. Addison-Wesley 2000.

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

[6] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice-Hall, 1994.

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

[8] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.

[9] C. B. Jones. *Tentative steps toward a development method for interfering programs.* ACM Transactions on Programming Languages and Systems, 5(4):596-619, 1983.

[10] 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 (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, 2000, pp. 513-527.

[11] C. Morgan. *Programming from Specifications*. Second Edition, Prentice Hall 1994.

[12] L. Petre, R-J. Back and I. Paltor. *Analysing UML Use Cases as Contracts*. Proceedings of UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, USA, October 1999.

[13] B. Potter, J. Sinclair and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.

[14] 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).

[15] S. Sendall and A. Strohmeier. *From Use Cases to System Operation Specifications.* 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 (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, 2000, pp. 1-15.

[16] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[17] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

10 Electronic Resources

[18] Klasse Objecten. OCL Center: OCL Tools. http://www.klasse.nl/ocl/index.htm

[19] OMG Unified Modeling Language Revision Task Force. OMG Unified Modeling Language Specification, Version 1.3, June 1999. http://www.celigent.com/omg/umlrtf/

[20] Presidents Information Technology Advisory Committee. Report to the President "Information Technology Research: Investing in Our Future". National Coordination Office for Computing, Information, and Communications, February 1999. http://www.ccic.gov/ac/report/pitac_report.pdf