From Use Cases to System Operation Specifications

Shane Sendall and Alfred Strohmeier

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

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

ABSTRACT The purpose of this paper is to first showcase the concept of an operation schema—a precise form of system-level operation specification—and secondly show how operation schemas enhance development when they are used as a supplement to use case descriptions. An operation schema declaratively describes the effects of a system operation by pre- and postconditions using the Object Constraint Language (OCL), as defined by the Unified Modeling Language (UML). In particular, the paper highlights techniques to map use cases to operation schemas and discusses the advantages of doing so in terms of clarifying the granularity and purpose of use cases and providing a precise specification of system behavior.

KEYWORDS Unified Modeling Language, Use Cases, Object Constraint Language, Operation Specification, Object-Oriented Software Development.

1 Introduction

The invasion of software-intensive systems into nearly every domain of our life has seen the practice of software development stretched to combat the ever-increasing complexity of such systems and to meet the increased demand. In such a development environment, the transformation from concept to running implementation needs to rapidly meet the market demand, but at the same time the software should exhibit the necessary qualities of robustness, maintainability, and meet other requirements, such as usability and performance demands. Currently, software is often lacking quality—as observed by the US President's IT Advisory Committee [10],

> "We have become dangerously dependent on large software systems whose behavior is not well understood and which often fail in unpredicted ways."

They continue further on, pin-pointing one of the deficiencies in current software development,

> "Having meaningful and standardized behavioral specifications would make it feasible to determine the properties of a software system and enable more thorough and less costly testing. Unfortunately such specifications are rarely used. Even less frequently is there a correspondence between a specification and the software itself."

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.

In this paper, we look at bringing the benefits of behavioral specification techniques to main-stream software development by proposing the use of operation schemas as a supplement to use cases. An operation schema declaratively describes the effects of a

system operation by pre- and postconditions using the Object Constraint Language (OCL) [16], as defined by the Unified Modeling Language (UML) [9]. We illustrate the advantages of complementing use cases with operation schemas by an example of a elevator control system. Moreover, we look at how one gets from use cases to operation schemas, and thus propose a mapping from use cases to operation schemas.

This paper is composed in the following way: section 2 gives an introduction to use cases; section 3 describes the elevator control system example that is used throughout the paper; section 4 gives some motivation for supplementing use cases with operation schemas; section 5 provides an introduction to operation schemas and OCL; section 6 proposes a mapping from a use case to operation schemas and demonstrates it for the elevator control system example; section 7 discusses related work; and finally section 8 concludes the paper and proposes possible future work.

2 Use Cases

Use cases are used to capture behavioral requirements of software systems. Use cases are popular because of their informal, easy to use and to understand style which caters to technical as well as non-technical stakeholders of the software under development. Use cases can be decomposed into further use cases, and therefore they are scalable to any system size. Moreover, it is possible to trace between subordinate use cases and the “parent” use case. Also, use cases have a wide spectrum of applicability: they can be used in many different ways, in many different domains, even non-software domains, making them a very versatile tool to have in one’s development toolkit.

Contrary to popular belief, use cases are primarily textual descriptions, the graphical appearance, called a use case diagram in UML, tells nothing more than the names of the use cases and their relationships to actors. This graphical appearance is just used to give an overview of the use cases in focus, from which allocation of work can be partitioned, for example.

UML also defines three relationships that can be used to structure use cases: “extend”, “include”, and “specialization”. These relationships help to avoid duplication of work and try to direct one towards a more object-oriented view of the world rather than towards functional decomposition. UML, however, does not go into detail about what content a use case description consists in and how it is structured. UML states that textual descriptions can be used, but that activity and state diagrams might be another alternative.

In practice, use cases can have varying degrees of formality depending on how much and what kind of information is recorded. Some will be just a casual story-telling description, others will be fully-dressed use cases that include an assortment of secondary information and that are possibly even described using pre- and postconditions. The type of template and style one chooses reflects how and where one is going to model with use cases.

On the one hand, the loose guidelines governing the general form of use cases has seen use cases used in new and imaginative ways, allowing many flavors of use cases to grow and diversify, but on the other hand, the lack of strict guidelines/style in the original definition has led to confusion among inexperienced users as to what the structure and purpose of a use case should be.

We use and advocate a style of use case proposed by Cockburn [1]. This style elaborates on the original work on use cases by Ivar Jacobson. Jacobson's definition of a use case introduces the notion of transaction [6]:

> "A use case is a sequence of transactions performed by a system, which yields an observable result of value for a particular actor."

where he defines a transaction as:

> "A transaction consists of a set of actions performed by a system. A transaction is invoked by a stimulus from an actor to the system, or by a timed trigger within the system."

Cockburn's definition [1] highlights that effective use cases are goal-based:

> "A use case is a description of the possible sequences of interaction between the system under discussion and external actors, related to the goal of one particular actor."

Cockburn also clarifies Jacobson's notion of transaction. He states that it consists of 4 steps [1]:

> "1. The primary actor sends request and data to the system; 2. The system validates the request and the data; 3. The system alters its internal state; 4. The system replies to the actor with the result".

A use case describes every possible situation that can arise when a user has a particular goal against the system. Each "possible situation" that arises is referred to as a scenario, and a use case can be considered as a collection of related scenarios.

Cockburn [1] provides some suggestions for defining granularity levels in use cases. He identified three different abstraction levels, in terms of the view of the system: summary level is the 50,000 feet perspective, user-goal level is the sea-level perspective, sub-function is the underwater perspective. Summary level use cases show the life-cycle sequencing of related goals; they act as a table of contents for lower-level use cases. User-goal level use cases describe the goal that the primary actor has in trying to do something. A user-goal level use case is usually done by one person, in one place, at one time, and the actor can normally go away happy as soon as this goal is completed. Subfunction level use cases are those required to carry out user goals, they are low-level and are generally the level of operation schemas or below. Therefore, as a general rule of thumb, we do not normally deal with sub-function use cases, we use operation schemas instead. User-goal level use cases are of greatest interest to us, and we will illustrate one in the next section on the elevator example.

3 Elevator Control System Example

For illustrating our approach, we will "develop" an elevator control system. The system controls only one elevator cabin, which travels between each floor of a building. There is a single button on each floor to call the lift. Inside the elevator cabin, there is a series of buttons, one for each floor. Requests are definitive, i.e., they cannot be cancelled, and they persist; thus they should eventually be serviced. The arrival of the cabin at a floor is detected by a sensor. The system may ask the elevator to go up, go down or stop. In this example, we assume that the elevator's braking distance is negligible. The system may ask the elevator to open its door. The system will receive a notification when the door is closed. This simulates the activity of letting people on and off at each floor. The door closes automatically after a predefined amount of time. However, neither this function of the elevator nor the protection associated with the door

closing (stopping it from squashing people) are part of the system to realize. The full worked example is available at [13].

The example illustrates a special situation where the actors have only very limited pre-defined usage protocols with the system. This is not always the case: for example, interaction with human actors is usually a lot more complex.

For this system, we could imagine a summary-level use case that describes the life cycle of the elevator. But for reasons of size, we concentrate instead on one user-goal level use case called Take Lift. Take Lift describes the activity of a user taking a lift from one floor to another. The use case description for Take Lift is shown in figure 1.

Use Case: Take Lift

Scope: Elevator Control System

Level: User Goal

Intention in Context: The User intents to go from one floor to another. The System has a single lift cabin that may service many users at any one time.

Primary Actor: User

Main Success Scenario:
1. User requests System for lift from a particular floor [->externalRequest].
2. System acknowledges User's request and commands the lift to pick up User.
3. System ascertains that lift has reached User's floor^{+} [->atFloor]; it commands lift to stop and open its Door, and informs User.
4. User enters lift and requests System to go to a floor as destination [->internalRequest].
5. System acknowledges User's request.
6. Door informs System of closure [->doorIsClosed]; System commands lift to go to destination floor.
7. System ascertains that lift has reached destination floor^{+} [->atFloor]; it commands lift to stop and open its Door, and informs User.
8. User exits lift.

Extensions:
2a. System determines that the lift is available at the User's floor with door open:
    2a.1. System requests the Door to stay open; use case continues at step 4.
2b. System determines that the lift is available at the User's floor with door closed:
    2b.1. System commands lift to open its Door; use case continues at step 4.
2c. System ascertains that the lift is currently unavailable:
    2c.1. System acknowledges User's request.
    2c.2. System ascertains that the lift has reached User's floor^{+} [->atFloor]:
    2c.3. System commands lift to stop and open its Door, and informs User; use case continues at step 4.
(4-7)||a. User requests System to go to a number of additional floors* [->internalRequest]:
    (4-7)||a.1. System acknowledges each additional request of User; use case continues from where it was interrupted.
4a. User does not make a request (or took the stairs)**:
    4a.1. Door informs System of closure; use case ends in failure.
4b. Door informs System of closure^{++} [->doorIsClosed]:
    4b.1. User requests System to go to a floor as destination [->internalRequest].
    4b.2. System acknowledges User's request, and commands lift to go to destination floor; use case continues at step 7.
5a. System determines that lift is already at destination floor:
    5a.1. System commands lift to open its Door, and informs User; use case continues at step 8.
8a. System determines that there are additional requests pending*: use case continues at step 6.

Notes:
* System cannot make difference between a single User making a number of requests and many Users each making a request
** undetectable by system
$^{+}$ Floor Sensor indicates each floor that is reached
$^{++}$ User has already entered lift (undetectable by system)

Fig. 1. Take Lift Use Case

The Take Lift use case description, figure 1, follows, more or less, the template and style recommended by Cockburn [1]. It consists of seven different sections, in which the Main success scenario and the Extensions sections describe the different steps of the use case. The Main success scenario section describes the standard path through the use case. The Extensions section describes the alternatives to the standard path scenario. Sometimes an alternative supersedes the main step, e.g. step 2a, and sometimes it might happen in addition to the main step, e.g. step 4 || (interleaved or in parallel). An alternative might correspond to regular behavior, exceptional behavior that is recoverable, or unrecoverable erroneous behavior.

The use case takes the user's perspective and, for example, differentiates "the user takes the stairs" from "the user enters the lift but does not make a request", whereas the system's viewpoint would not do so.

Now taking a look at the interactions between the system and the actors which realize the Take Lift use case, we see in figure 2 the system interface necessary for the use case. In section 6, we will discuss how we derive these events and the corresponding operations that they invoke from the Take Lift use case, but for the moment we can just assume their existence. The System Context Model, figure 2, 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.

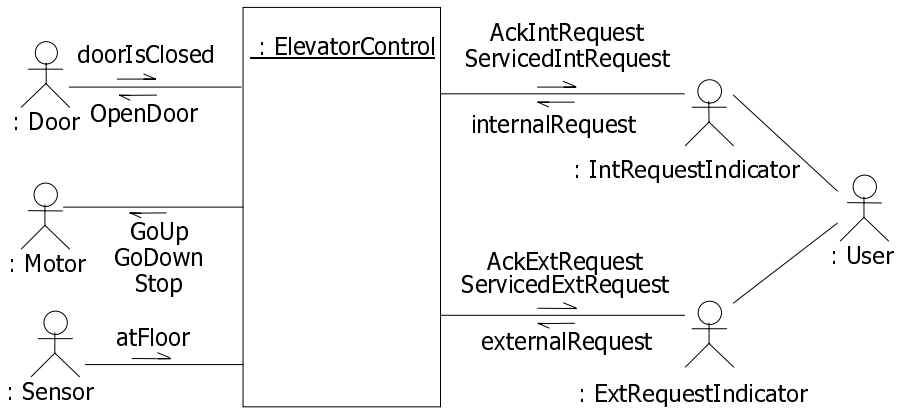

Fig. 2. Elevator Control System Context Model

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 3. It shows concepts of the domain and the relationships between them. Inside the system there are three classes, Cabin, Floor, and Request, and outside six actor classes, Motor, Door, IntRequestIndicator, ExtRequestIndicator, User, and Sensor. Note that these are not design objects but rather entities used to represent the system state. 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.

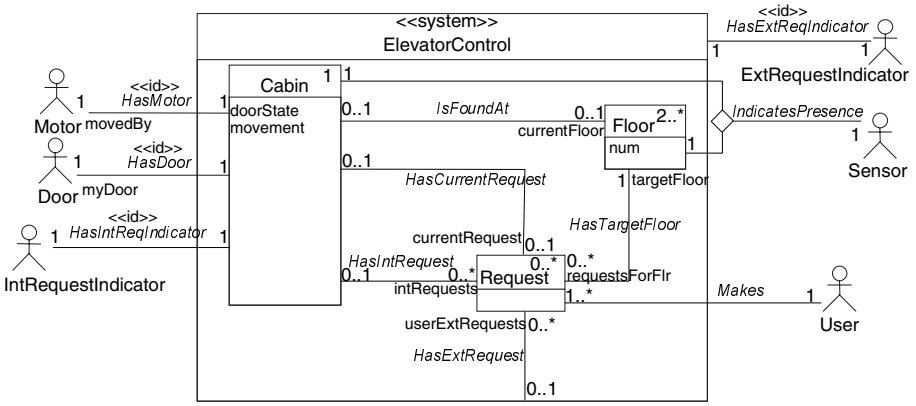

Fig. 3. Elevator Control System Class Model

4 Supplementing Use Cases with Operation Schemas

Use cases are an excellent tool for capturing behavioral requirements because they describe the system from a user's point of view, which naturally allows one to describe not only normal behavior—successful scenarios—but also abnormal behavior—unsuccessful and exceptional scenarios. Another approach, or view on the system is where one looks purely at the interface and functionality offered by the system. This is the view provided by operation schemas.

The two views complement each other nicely: use cases provide the informal map of interactions between the system and actors, whereas operation schemas precisely describe a particular system action which executes atomically, called a system operation. A system operation corresponds to a transaction as defined by Jacobson (see section 2), a sequence of which forms a use case.

Operation schemas complement use cases in a number of ways. Firstly, use cases face the usual problems of descriptions written in natural language, i.e., they may be ambiguous and their level of description may vary, making it easy to fall into a design description. However, operation schemas are less likely to have such problems due to the use of OCL and are less likely to embody premature design decisions due to this single level of description. Secondly, operation schemas being precise and more formal than natural language offer a more rigorous basis on which we can reason about system properties, verify that invariants are obeyed, and provide a basis for specification-based testing. Potentially much of the verification and testing can be automated because of the formal nature of OCL. Thirdly, finding the right granularity for a use case is difficult, because there is a danger of decomposing use cases into pieces too small. On the contrary, operation schemas break the recursive decomposition at the

level of operation schemas. Finally, operation schemas have a one-to-one mapping to collaboration diagrams, an important design artifact. One operation schema is realized by one collaboration diagram. A use case, on the other hand, does not map well to any single design artifact. As Cockburn [1] observed in the context of use cases:

"Design doesn't cluster by use case, and blindly following the use case structure leads to functional decomposition design".

It is worthwhile to note that operation schemas do not offer any advice on the allocation of behavior to objects: all the work is still to be done in terms of designing the objects of the system. The integration of collaboration diagrams into a coherent architecture is outside the scope of this paper, but interested readers are referred to [8].

We have found the approach of taking use cases to operation schemas enhances the development of reactive systems. However, we do acknowledge introducing operation schemas into a project requires an upfront cost for learning OCL and it forces one to spend longer in conceptual phases of development, which are often perceived by management as the "non-productive" phases.

5 Operation Schemas

An operation schema describes the effect of the operation on an abstract state representation of the system and any 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 observed after the execution of the operation by a postcondition. Operation schemas use UML's OCL formalism.

The system model is reactive in nature and all communication with the environment is achieved by asynchronous input/output events. 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 a class model. 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 4. 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 the declarations are in line with the proposal of Cook et al. [4].

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. 4. Operation Schema Format

5.1 Presentation of OCL

UML [11] defines a navigation language called the Object Constraint Language (OCL) [16], a formal language 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.

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.

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 the 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.

```
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;
Use Cases: take lift;
Scope: Cabin; Floor; Request; 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 Elevator Control System

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.

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.

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, but deletes also all the association links connected to the deleted objects. This sort of implicit removal ensures consistency of associations.

Also, the & operator used throughout the schema is a shorthand for logical "and". 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. 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.

For a discussion on the syntax and semantics of operation schemas see [12] [15].

6 Mapping Use Cases to Operation Schemas

In this section, we describe the activity of deriving system operations from use cases. Generally, operation schemas are derived from user-goal level use cases, but sometimes sub-function level use cases can be useful too. The general rule is to decompose use cases into sub-use cases until we get step in the Main success scenario of the use case that have the granularity of a system operation, more or less.

This mapping activity is not necessarily straight-forward because the interaction with secondary actors can often be vague and may require further clarification of the use case in question. Before this mapping activity is started, a class model is made for the system, or may already exist, which will be used to describe the system state for writing the operation schemas. This class model (see figure 3 for an example) gets refined as the operation schemas are worked out. Moreover, the mapping activity generally requires iterations of refinement between the models.

The general approach for mapping a use case to its corresponding operation schemas involves analyzing each step of the use case, looking for events sent by actors that trigger a system operation. Once a trigger has been found, the system operation is described by an operation schema. The ultimate goal of the mapping activity is to partition the use case into a sequence of system operations. An important point to remem-

ber when deriving system operations from the use case is that for each system operation a triggering event must be found. Once we have decomposed the use case into system operations, we stop decomposition. Indeed, we have found that further decomposition often leads to structured design instead of object-oriented design.

We now show, stage-by-stage, the derivation of system operations from the Take Lift use case (figure 1). Step 1 of the Main success scenario describes the user making an external request for a lift—this is a trigger for a system action. The system action comes in step 2 where the system commands the lift to service the request. We, therefore, define a system operation called externalRequest which handles this action and specify it in an operation schema. To make the trace explicit, we add a hyperlink from the use case to the operation schema[1] (see the end of step 1 in figure 1).

Looking now at step 3, we see that the system performs an action, and we need, therefore, to find a trigger for it. Looking closely at the text, we see that the situation was brought up by the lift reaching the source floor. We look back to the original problem description (section 3) and see that we receive an event whenever the lift reaches a certain floor, in our case the source floor. We, therefore, define a system operation called atFloor, triggered by the floor sensor, which handles the action of stopping the lift, opening the door and dismissing the request, and specify it in an operation schema.

Looking at the step 4, it indicates a request to the system, except this time the request is from inside the lift. We, therefore, define a system operation called internalRequest which handles this action and specify it in an operation schema. Step 5 indicates that the system action corresponding to the request. In the main success scenario, the door closes after the user makes a request; in this case, the door closing event is used to trigger the movement of the lift to the destination floor. We, therefore, define a system operation called doorIsClosed, triggered by the door, which is responsible for the action of calculating and handling where the lift is to go next, and specify it in an operation schema.

A summary of all the event exchanges between the system and the actors is shown by the System Context Model in figure 2. This model indicates that there are four kinds of input events: externalRequest, doorIsClosed, atFloor and internalRequest. These input events trigger the system operations of the same names. We have already discussed the operation schema for the atFloor system operation. For reason of size, the other system operation schemas are not shown. Interested readers, however, can find them on the web [13].

7 Related Work

The idea of operation schema descriptions comes from the work on the Fusion method by Coleman et al. [2]. 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. After the initial work on Fusion, Coleman introduced use cases into Fusion and briefly discussed the relationship between them and opera-

1. Our convention for referencing the operation schema is to put the system operation name (hyperlinked) inside square brackets with an arrow preceding the name, e.g. [-> sysOpX].

tion schemas [3]. Our work on relating operation schemas to use cases can be seen as an elaboration of this work.

Z [14] and VDM [7] 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 [5], 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, and it was for this reason that we chose to supplement use cases with operation schemas rather than replace them with operation schemas. In addition, effective use cases have the ability to describe complex sequencing in a understandable and intuitive way; more formal approaches, such as joint actions, in the presence of complex sequencing are less intuitive to understand and can be hard to produce due to inflexibility of formal languages.

8 Conclusion

We described an approach that supplements use case descriptions with operation schemas. An operation schema is a declarative specification of a system operation written in OCL. We believe that we have shown that supplementing use cases with operation schemas provides the benefits of rigorous behavioral specification while still retaining the advantages of goal-based use cases. Moreover, we highlighted a possible mapping between a use case and its corresponding operation schemas on an elevator control system example.

Currently, we are focusing our work on the description of *concurrent* system operations and on the development of tool support for operation schemas.

References

[1] A. Cockburn. *Writing Effective Use Cases.* Addison-Wesley 2000.

[2] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall 1994.

[3] D. Coleman. *Fusion with Use Cases - Extending Fusion for Requirements Modelling*. OOPSLA Conference Tutorial Slides 1995.

[4] S. Cook, A. Kleppe, R. Mitchell, J. Warmer, A. Wills. *Defining the Context of OCL Expressions*. Second International Conference on the Unified Modeling Language: UML'99, Fort Collins, USA, 1999.
[5] D. D'Souza and A.Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley 1998.
[6] I. Jacobson, M. Griss and P. Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*. Addison-Wesley 1997.
[7] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.
[8] M. Kandé and A. Strohmeier. Towards a UML Profile for Software Architecture. Technical Report 2000, Swiss Federal Institute of Technology, Switzerland, 2000; submitted for publication.
[9] OMG Unified Modeling Language Specification, Version 1.3, June 1999; published by the OMG Unified Modeling Language Revision Task Force on its WEB site: http://www.celigent.com/omg/umlrtf/
[10] 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).
[11] J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley 1999.
[12] S. Sendall and A. Strohmeier. *Using OCL and UML to Specify System Behavior*. Technical Report EPFL-DI No 01/359, 2001.
[13] S. Sendall. *Specification Case Studies*. Electronic Resource: http://lglwww.epfl.ch/~sendall/case-studies
[14] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.
[15] A. Strohmeier and S. Sendall. Operation Schemas and OCL. Technical Report EPFL-DI No 01/358, 2001
[16] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley 1998.