

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 post conditions 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 post conditions using the Object Constraint Language (OCL) [15], 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 multi-cabin 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: "extends", "includes", 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 post conditions. 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, and 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 carrying 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 multiple lift cabins that all service the same floors of a building. There are buttons to go up and down on each floor to request the lift (apart from the top-most and bottom-most floors). Inside an elevator cabin, there is a series of buttons, one for each floor.

The arrival of a 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 and it receives a notification when the door is closed--the door closes automatically after a predefined amount of time, which simulates the activity of letting people on and off at each floor. However, neither this function of the elevator nor the protections associated with the door closing (stopping it from squashing people) are part of the system to realize. Finally, for reasons of conciseness in this paper, we have removed the capability of canceling requests, therefore a request is definitive. The full worked example is available at [13].

The example illustrates a special situation where the actors have only very limited predefined 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.

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 7 || (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`, `doorsClosed`, 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 a showing external communication path often clarifies the consistent overall working of a system.

Use Case: Take Lift**Scope:** "System" means the Multi-Cabin Elevator Control System**Level:** User goal**Goal:** A User wants to go from one floor to another.**Context of Use:** The Lift system has many lift cabins that service many users at any one time, taking them from one floor to another. Primary Actor: User**Main Success Scenario:**

1. The User requests a lift from a particular floor (source floor), also indicating the direction desired.
 2. The System acknowledges the external request and commands the most suitable lift (which is currently idle) to go to the source floor [-> externalRequest].
 3. The lift has reached the source floor, the System commands the lift to stop and open its Door, and the System dismisses the original external request of the User [-> atFloor].
 4. The User enters the lift.
- Steps 5 and 6 can happen in parallel (also implying any order)*
5. The lift door closes.
 6. The User requests a destination floor (inside the cabin).
 7. The System acknowledges the internal request and commands the lift to go to the destination floor [-> internalRequest].
 8. The lift has reached the destination floor, the System commands the lift to stop and open its Door, and the System dismisses the internal request of the User [-> atFloor].
 9. The User exits the lift.

Extensions:

- 2a. A lift is already at the source floor with the door open.
 - 2a.1. The System acknowledges the external request and dismisses it [-> externalRequest]; and the use case continues at step 4.
- 2b. A lift is already at the source floor with the door closed and is currently idle (not servicing another request).
 - 2b.1. The System acknowledges the external request, commands the lift to open its Door, and dismisses the external request [-> externalRequest]; the use case continues at step 4.
- 2c. No lifts are currently available.
 - 2c.1. The System acknowledges and schedules the request [-> externalRequest].
 - 2c.2. A lift becomes available.
 - 2c.2.1 The System commands the lift to go to the destination floor [-> doorsClosed]; continues step 3. (2-3)|. The User leaves and takes the stairs: the use case ends after step 3.
- 3a. The lift never reaches the source floor or the Door does not open:
 - 3a.1 .The User unhappily takes the stairs; the use case ends.
- 6a. The User does not make a request: the use case ends.
- 6b. The User(s) requests several different floors.
 - 6b.1. The System schedules all the internal requests, acknowledges them and commands the lift to go to the first floor that was requested [-> internalRequest].
 - 6b.2. The lift stops off at each floor requested, dismissing the requests on the way [-> atFloor] [-> doorsClosed]; the user may exit the lift at any stop off; the use cases ends.
- 6c. The User requests a destination floor (inside the cabin), but the lift door is still open.
 - 6c.1. The System acknowledges and schedules the request [-> internalRequest].
 - 6c.2. The lift door closes: The System commands the lift to go to the destination floor [-> doorsClosed]; the use case continues at step 8.
- 7|. The User requests another destination.
 - 7|.1. The System schedules the request, in addition to the current request [-> internalRequest].
 - 7|.2. same as 6.b.2
- 7a. The lift is already at the destination floor.
 - 7a.1. The System commands the lift to reopen its Door, and the System dismisses the internal request of the User [-> internalRequest]; the use case continues at step 9.
- 8a. The lift does not drop off the User at the destination floor, either because the Door doesn't open or the lift never reaches the destination floor.
 - 8a.1 .The User, by mobile phone, sues the company who made the multi-cabin elevator system (not automated); the use case ends.

Fig. 1. Take Lift Use Case

The analysis-level class model for the elevator control system is shown in figure 3. 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`.

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, which was built with the purpose to be writable and readable by developers.

The system model is reactive in nature and all communication with the environment is achieved by asynchronous input/output events, termed signals in UML¹. 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 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 additional comments.

Use Cases: This clause contains cross-references to super ordinate use case(s).

Scope: This clause declares the classes, associations, and system-wide objects of the class model that are used in the schema.

Declares: This clause provides declarations of all constants and variables designating objects, data types, object collections, and data type collections used in the Pre and Post clauses.

Sends: This clause specifies which kinds of events are sent to which actor types. Sending an event is modeled by placing the event into the local event queue of the destination actor. It is also possible to declare event instances and event collections, and to enforce event sequencing.

Pre: This clause is the operation's precondition written in OCL that evaluates to true or false. The precondition is made up of possibly many Boolean OCL expressions separated by semi-colons (;). A semi-colon replaces a "logical and" and is used as a Boolean expression *terminator* (not separator).

Post: This clause is the operation's post condition. Like the precondition, the post condition is made up of possibly many Boolean OCL expressions separated by semi-colons (;). If the precondition is true, then the post condition 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 can refer to the pre-state, by the notation *~pre*.

Fig. 4. Operation Schema Format

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

5.1 Presentation of OCL

UML [11] defines a navigation language called the Object Constraint Language (OCL) [15], 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 post conditions 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.

We now highlight the `atFloor` operation schema, shown in figure 5. The `atFloor` operation schema describes the action of the system to stop at a particular floor or to continue on to the next one; the decision is based solely on whether there are any requests for the floor. The operation is triggered by a floor sensor when it detects the cabin at a particular floor. For the moment, we will ignore how we derived this operation schema from the Take Lift use case and just concentrate on the syntax and content of the schema itself.

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 the event types `Stop`, `GoUp`, `GoDown`, `Open Door`, `ServicedExtRequest`, `ServicedIntRequest` are in scope and that `Stop` and `OpenDoor` have named instances. Finally, it states that the two event instances are sent in the order `stop` followed by `open`. The **Pre** clause states that the cabin `cab` has a `currentRequest`, i.e., `cab` is 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`.

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 and
```

```
    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;
```

```
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. 5. atFloor Operation Schema for Elevator Control System

The first line of the Post clause states that the cabin is now found at floor f. The next (compound) expression states that if a stop was made, 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. For an explanation of the frame assumption for operation schemas which explains this sort of implicit removal, see [12].

In the **Post** clause, sending events is described by stating that an event instance was put into the event queue of 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 an event instance called `stop` in its local event queue. Looking closer 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.

In the description of postconditions we use the principle of minimum set [12] to clarify the semantics of the Post clause, to make postconditions more readable and to make it possible to state postconditions incrementally. For example, if `request1` and `request5` are linked to a cabin, `cab`, via the `HasIntRequest` association, then the following two conditions are equivalent:

```
cab.intRequests->includes (request1) and cab.intRequests-> includes (request5);      (1)
cab.intRequests =cab.intRequests@pre->union (Set {request1, request5})           (2)
```

The minimum set principle applied to the first condition guarantees that no extra elements are included in the `cab.intRequests` set after the execution of the operation (other than `request1` and `request5`).

For a discussion on the semantic aspects of operation schemas see [12].

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 role is to decompose use cases until we get to use cases, where each step of the use case is 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 which is a first approximation of the "system state", which will be needed to write 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 exhibits iterations of refining the use case and writing/updating the corresponding operation schemas and the class model.

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 remember 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 a 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 schema¹ (see the end of step 2 in figure 1).

Looking now at step 3, we see that the system performs an action, and we need, therefore, to trigger 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 steps 4 and 5, we see that no obligation is placed on the system to do anything (although note that we will receive an event from the door informing us that the door is closed). Similarly to the steps 1 and 2, the steps 6 and 7 indicate a request and a corresponding system action, 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. Similarly to step 3, step 8 is handled by the `atFloor` system operation. And looking at step 9, we see that no event or obligation is placed on the system.

Now let's look at the steps in the Extensions clause. The steps 2a and 2b, alternatives to step 2, detail other situations that need to be dealt with in the `externalRequest` system operation. Looking at step 2c, we have a condition that no lifts are available; this means that the `externalRequest` system operation is responsible for step 2.c.1, i.e., acknowledging and queuing the request. However, we see in step 2.c.2.1 that the system performs an action, and again we need a trigger. Examining closely the text, we see that a lift is in a position to go to another floor, in this case to do a pick up. We look back to the original problem description (section 3) and see that we receive an event whenever the lift closes its door. This event seems to be the best candidate for moving off the lift. We, therefore, define a system operation called `doorsClosed`, 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.

Step (2-3)|| means that the step can be done in parallel to the steps 2 or 3. This step along with steps 3a and 6a have no effect on the system functionality. The steps 6b and 7|| involve multiple stop offs by the lift, and therefore they queue the request with the `internalRequest` system operation and use a combination of `atFloor` and `doorsClosed` system operations for stopping and moving off the lift, respectively. The step 7b details a situation that needs to be dealt with in the `internalRequest` system operation. Finally, the step 8a has no effect on the system functionality.

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].

+

Operation: ElevatorControl::externalRequest (f: Floor, d: Direction)
Description: An external request to get a lift, indicating the direction desired, is made.
Notes: The system can receive many externalRequest input events at any one time.
Use Case(s): TakeLift;
Scope: Cabin; HasCurrentRequest; IsFoundAt; HasExtRequest; HasTargetFloor; ExtRequest; Floor;
Declares:
alreadyRequested: Boolean ::= self.extRequest->exists (r | r. direction = d and
r. targetFloor = f);
allAvailableCabins: Set (Cabin) ::= self.cabin->select (c | ((c.currentRequest->isEmpty and
c.doorState = #closed) or (c.currentFloor = f and c.doorState = #open))));
-- the set of all cabins that are idle, or stopped with door open at the right floor
cab: Cabin;
req: ExtRequest;
Sends:
Door::{OpenDoor}, Motor::{GoUp, GoDown},
ExtRequestIndicator::{AckExtRequest, ServicedExtRequest};
Pre:
true
Post:
If not alreadyRequested then
if allAvailableCabins->notEmpty **then** -- if cabins are available
cab = bestSuitedCabin (allAvailableCabins, f); -- cab is best suited cabin
if cab.currentFloor@pre = f **then** -- cab is currently at the requested floor
cab.doorState = #open;
(cab.myDoor).events->includes (OpenDoor' ()); -- note: door may have already been open
(self.extReqindicator).events->includes (ServicedExtRequest' (callingFlr => f,
dir => d));
else -- the lift was not on floor f
req.ocllsNew (direction => d);
req.targetFloor = f;
self.extRequest->includes (req);
cab.currentRequest = req;
if (f.num > cab.currentFloor@pre.num) **then** -- if lift is to go up
cab.movement = #up;
(cab.movedBy).events->includes (GoUp' ());
else -- if lift is to go down
cab.movement = #down;
(cab.movedBy).events->includes (GoDown' ());
endif;
endif;
else -- else retain the request to deal with later when there is a lift that becomes free req.ocllsNew (direction => d);
req.targetFloor = f;
self.extRequest->includes (req);
endif;
(self.extReqIndicator).events->includes (AckExtRequest' ());
endif;

Fig. 6. externalRequest Operation Schema for Elevator Control System

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, doorsClosed, 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. We will now present the operation schema for the externalRequest system operation. For reason of size, the other system operation schemas are not shown. Interested readers, however, can find them on the web [13].

The parameterized predicate bestSuitedCabin (line 3) results in the lift cabin that is best suited for servicing the request; we do not provide the definition of this parameterized predicate in this paper; it would be defined by taking into account the cabins' scheduling policy.

The Post clause also shows a convention for sending unnamed event instances. The condition, (cab.movedBy).events->includes (GoUp' ()), asserts that an unnamed event instance of the event type GoUp was put into the event queue of the actor instance cab.movedBy.

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 operation 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^o 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 roles, 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. On the other hand, it could be argued against our approach that refining use cases to operation schemas is less direct and more heuristic-driven because of the gap between the two notations.

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. Kand6 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://uml.shl.com/artifacts.htm>
- [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/pitacreport.pdf>).
- [11] J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley 1999.
- [12] S. Sendall and A. Strohmeier. *Descriptive Object-Oriented Operation Specification for UML*. Technical Report 2000/326, Swiss Federal Institute of Technology, Switzerland, 2000.
- [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] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley 1998.