Electronic Notes in Theoretical Computer Science 80 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume80.html 18 pages

Applying Fondue to Specify a Drink Vending Machine

Alfred Strohmeier, Thomas Baar [1]

Software Engineering Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL)
CH-1015 Lausanne, Switzerland

Shane Sendall [2]

Software Modeling and Verification Laboratory
University of Geneva, Computer Science Department
24, rue du Général-Dufour, CH-1211 Genève 4, Switzerland

Abstract

The purpose of the paper is to present our approach for specifying system behavior during analysis, part of the Fondue software development method. The approach is exemplified on a case study, a Drink Vending Machine (DVM). It is based on Operation Schemas and a Protocol Model. The Protocol Model describes the temporal ordering of the system operations by an UML protocol statemachine. An Operation Schema describes the functionality of a system operation by pre- and postconditions; they are written in the Object Constraint Language (OCL), with a few amendments and extensions.

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.

Key words: Object-Oriented Software Development; Software Development Method; Software Specification; Formal Specification; Unified Modeling Language (UML); Object Constraint Language (OCL); Fondue Software Development Method.

[1] Email: [Alfred.Strohmeier|Thomas.Baar]@epfl.ch
[2] Email: Shane.Sendall@cui.unige.ch

©2003 Published by Elsevier Science B. V.

1 Introduction

UML and OCL

UML [7] 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 [2]. 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 [12] and VDM [5] 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.

Formal methods, like Z and VDM, suffer from the problem that they are very costly to introduce into software development environments, because of their high requirements for mathematical maturity on the user. On the other hand, OCL, part of the UML [7], 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.

Fondue

Fondue is an object-oriented software development method developed by the Software Engineering Lab of EPFL. Fondue covers in a consistent approach all phases from requirements elicitation, and analysis, over design to implementation. As we will see, it leads from requirements based on Use Cases to system operation specifications using pre- and postconditions written in OCL.

The Fondue method was first described in [10], and then enhanced by the addition of a requirements elicitation activity [11]. The Fondue analysis phase, together with more advanced, somewhat speculative, material was documented in a Ph.D. thesis [9].

Fondue has its origins in the well-known Fusion method [3]; it adopts its process, but uses the UML notations. In addition to Fusion, Use Cases are

proposed for requirements elicitation and are taken into account during the analysis phase. The Fondue method not only provides an internal view of the class model and the behavior of individual classes, but it includes modeling of system-wide functionality and a step-by-step process that leads the development team from an initial requirements document through to the implementation of an object-oriented software system. Fondue defines a number of deliverables, one of which defines a specification of system behavior. The specification includes three principal views [10], see Fig. 1.

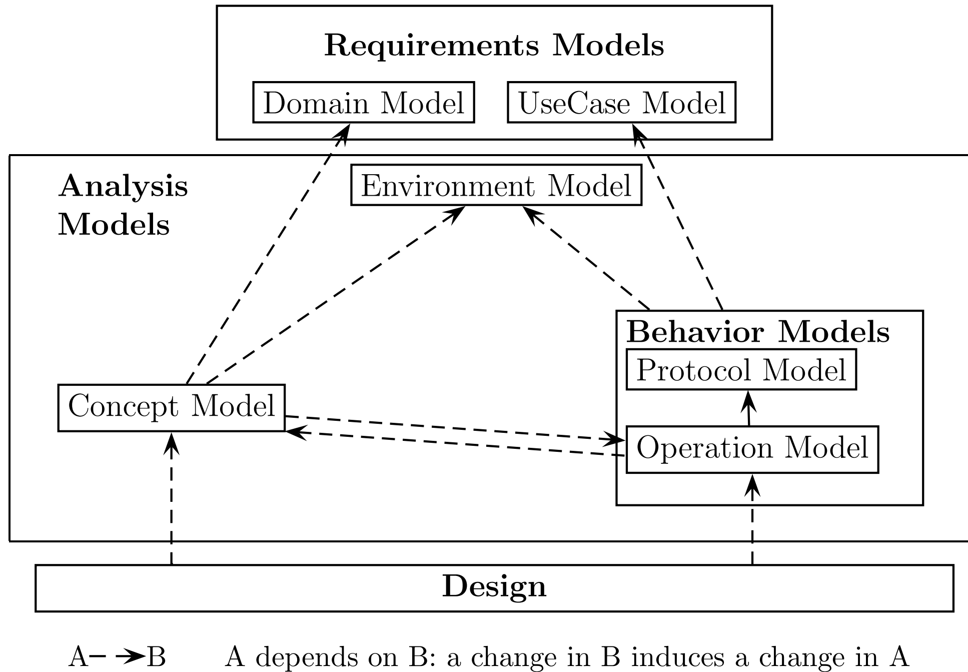

Fig. 1. Fondue Specification Models

- Operation Model: The Operation Model is composed of Operation Schemas, one for each system operation. An operation schema describes by pre- and postcondition assertions written in OCL the effects on system state and messages output by the system caused by the execution of an operation;
- Protocol Model: a restricted form of UML statechart that defines the allowable temporal ordering of operations; and
- Concept Model: a description, constructed using the UML class diagram notation, that defines the system state that is required to describe the effects of operation executions in the Operation Schemas.

The underlying system model is reactive in nature and all communication with the environment is achieved by sending messages. Upon reception, a message triggers an asynchronous event, which might eventually trigger a system operation.

Environment Model

Message signatures and, more importantly, the exchange of messages with actors belonging to the system´s surroundings is shown in an ancillary model, called the Environment Model.

Concept Model

The Concept Model, a special-purpose class model, is used to describe all the concepts and relationships part of 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.

Operation Model

An Operation Schema describes the effect of the operation on an abstract state representation of the system and by messages 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 and instantaneously.

All system operations are triggered by input events, normally of the same name as the triggering message and 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 Concept 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 messages 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.

Protocol Model

A Protocol Model is a UML protocol statemachine that focuses on the temporal ordering of the system operations only, and therefore the usage of the UML statechart 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 Protocol Model describes the allowable sequencing of these services.

Behavior Model

Both the Operation Model and the Protocol Model are refined from Use Cases and they combine to define a precise specification of system behavior, the Behavior Model. An approach for mapping Use Cases to Operation Schemas has been proposed in [11]. To see how this work fits into a software development analysis activity, the reader is referred to [10].

Case Study

The use of the Fondue method for specifying a system will be showcased on a Drink Vending Machine. This machine corresponds to a typical, but simple, reactive system. It is composed by a controller, whose software has to be developed, and several peripheral hardware devices. These devices interact with the software system by exchanging asynchronous messages. Ordering constraints for the messages are specified by the Protocol Model, whereas the effect of receiving a message is specified by an Operation Schema.

Structure of the paper

We start in Section 2 by presenting informally the problem, that is the Drink Vending Machine. The typical usages of such a machine are described in Section 3 by Use Cases. In Section 4 we start elaborating the corresponding Fondue specification by modeling the environment of the Drink Vending Machine. Section 5 provides its Concept Model. Allowable sequences of operations performed on the Drink Vending Machine are shown by the Protocol Model in Section 6. The Operation Model of Section 7 completes the specification. Section 8 states the lessons learnt and draws some conclusions.

2 Original Problem Statement

When using a Drink Vending Machine the following steps typically are taken:

(i) the consumer introduces coins in the machine,

(ii) s/he selects the desired drink, and

(iii) the dispenser supplies the drink via the bottom drawer.

One has to distinguish between three levels: the physical machine, the controller, and human interaction. The physical machine consists of several components: a money box, drink shelves, an information panel, drink selection buttons, and an eject coins button. The controller is a program that coordinates the activities of the system´s components by receiving messages from and sending commands to them. Human interaction takes place between the consumer or service person and the system´s components, for example, a consumer presses a drink selection button or the service person replenishes a shelf.

The money box is composed of two different collectors of coins: the first one keeps coins until the consumer presses the eject button or chooses a drink; in the latter case the coins are released by the first collector to the second one.

Drinks are stored on shelves. Each shelf is associated with a beverage kind and a price. The service person adds drinks to the shelves. Initially, there are no drinks. The system has some constraints:

- the consumer can only insert coins up to a certain limit greater than the highest drink price; any additional coin will be released immediately by the first collector (in order to enforce the consumer to choose a drink or to press the eject button),
- the money box (second collector) has limited capacity,
- the number of shelves that the machine possesses is fixed (i.e., constant throughout the life of the system),
- each shelf is subdivided in slots,
- there is a maximum of one drink per slot,
- all shelves have the same number of slots and their number is fixed for the life of the system.

3 Use Cases

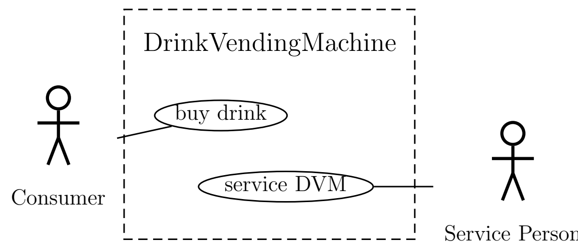

Fig. 2. Use Case Diagram of DrinkVendingMachine

The Use Case diagram shown in Fig. 2 is very simple. The essential information is kept in the Use Case *descriptions*. The Fondue format to describe Use Cases is similar to the format proposed by Cockburn [2].

As usual, during analysis it was discovered that the problem statement, including the Use Cases, was imprecise and incomplete. We provide here some additional rules:

(i) When the money box (second collector) is full, then the DVM gets out of order.

(ii) As long as the DVM is not out of order, it is able to collect the money needed for buying a drink. Otherwise stated, the money box (second collector) is either full, or it has enough capacity to collect the money due for another drink.

(iii) When a shelf delivers its last drink, it reports that it became empty.

(iv) The DVM does not have a clever money system. One might e.g. suppose that it accepts only one kind of coins, e.g. quarters, and that all prices are multiples of the value of that coin, e.g. 25 cents. Also, it is then easy to return coins that are too many compared with the price.

3.1 Buy Drink

Use Case: buy drink

Scope: Drink Vending Machine

Level: User Goal, Black-box

Intention in Context: The intention of the Consumer is to buy a drink from the machine. This involves the exchange of a certain amount of money for a drink.

Primary Actor: Consumer

Precondition: System must be initialized (where a shelf has a known price).

Main Success Scenario:

1. Consumer introduces a coin into the machine.

Step 1 can be repeated as many times as the Consumer wishes.

2. Consumer selects a drink.
3. System validates that there are sufficient funds for the selection, takes the specified amount of money (returning the excess amount of money to Consumer)* and dispense drink.
4. Consumer obtains drink from machine.

Extensions:

1a. DVM is out of order:
 1a.1. System releases inserted coin;
 1a.2. Consumer collects money; Use Case ends in failure

(1-2)a. Consumer requests ejection of money:
 (1-2)a.1. System ejects money.
 (1-2)a.2. Consumer collects money; Use Case ends in failure.

3a. System determines that there are insufficient funds for the purchase:
 3a.1. System informs Consumer; Use Case continues at step 1 or 2.

3b. System determines that there are no drinks left in that category:
 3b.1. System informs Consumer ; Use Case continues at step 1 or 2.

4 || a. Money box informs System that it is full:
 4 || a.1. System informs Consumer that it is out of order**;
  Use Case ends.

Notes: * We need to be careful that the Consumer does not request to eject the money after selecting a drink, but before the system has taken the money, i.e., so that the Consumer does not succeed in getting his/her money

back as well as a drink.
** The System stays out of order until reset by a Service Person.

3.2 Service DVM

Use Case: service DVM

Scope: Drink Vending Machine

Level: User Goal, Black-box

Intention in Context: The intention of the Service Person is to maintain the system by ensuring that the machine has drinks available and by collecting the money earned.

Primary Actor: Service Person

Main Success Scenario:

1. Service Person adds drinks to a shelf.
2. Service Person sets or changes the price of drinks on a shelf.
   Steps 1 and 2 are repeated for each shelf, in any order.
3. Service Person collects money earned from machine.

Extensions: – None

Notes: Steps 1 and 2 constitute the initialization of the system.

4 Environment Model

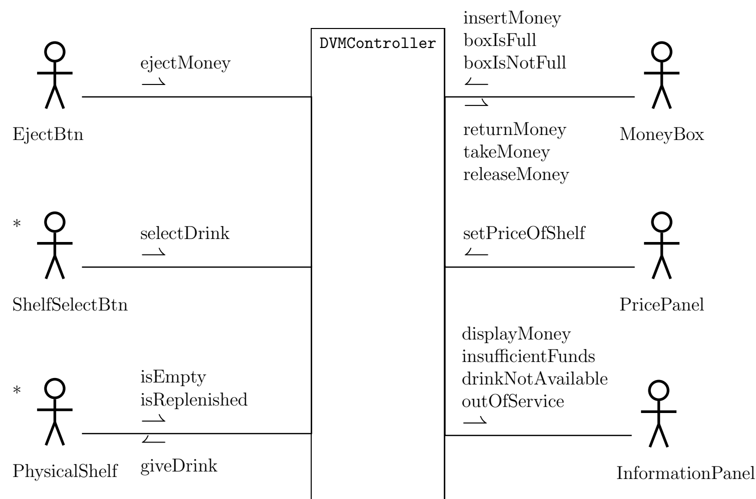

Fig. 3. Environment Model of DVMController

The Use Cases provide a structured but yet informal description of the system behavior. In order to get a formal behavior description, we first have to identify the subcomponents of the system and how they communicate among each other. As the next step, Fondue proposes therefore the development of an environment model as in Fig. 3.

The Environment Model identifies all messages the system sends to and receives from the environment. The environment is modeled by actors. Note, that Fondue allows (in opposite to official UML) actors to be decorated by a star (*PhysicalShelf*, *ShelfSelectBtn*). This indicates that more than one instance of the actor can interact with the system.

It might be a surprise to have different actors in the Environment Model and the Use Case diagram. The confusion disappears, once it is realized that we are mainly interested in the software part of the Drink Vending Machine, i.e. the controller that coordinates the activities of the system´s components. The coordination is achieved by sending messages between the controller and the components. From the viewpoint of the controller, the system´s components belong to the environment. The following question may arise: If the system´s components constitute the environment, where do the actors *Consumer* and *Service Person* used in the Use Cases belong to? In Fondue they do not play any further role. They interact with the DVM by pressing the eject button, inserting money into the money box, etc. However, for the DVMController, it is not relevant how the human interaction with the system´s components is performed, i.e. how the Consumer presses the button. Relevance for the controller has only the result of an interaction, e.g. the fact that the button was pressed.

The Environment Model given in Fig. 3 lacks some information which are both necessary and helpful within the next development steps. For the sake of brevity, the arguments of a message are suppressed in the diagram. They are given in the next table together with an informal description of actors and messages. This detailed description facilitates the understanding of the formal specification in the next Sections.

EjectBtn	button for ejecting all coins which are currently inserted into the first collector of the money box
ejectMoney()	the eject button has been pressed
ShelfSelectBtn	button for selecting a drink shelf; each ShelfSelectBtn corresponds to exactly one PhysicalShelf (see Concept Model in Section 5)
selectDrink()	the select button has been pressed
PhysicalShelf	physical drink shelf; is divided into slots and stores the drinks
isEmpty()	all slots are empty; Assumption: isEmpty() occurs after the last available drink was dispensed
isReplenished()	the shelf was replenished and is not empty anymore

giveDrink()	request from DVMController to dispense one drink
MoneyBox	consists of two collectors and a money return compartment; when the first collector is full, additional coins fall through; when the capacity of the second collector is exceeded, the DVM waits for the Service Person to exhaust the second collector
insertMoney(m:Money)	a coin of amount *m* was inserted (while the first collector was not full)
boxIsFull()	the second collector became full, i.e. there is no capacity left for paying another drink (occurs after collecting for good the money for a drink)
boxIsNotFull()	the second collector has again enough capacity due for another drink; boxIsNotFull() occurs after the Service Person has exhausted the second collector
returnMoney(m:Money)	request from DVMController to eject the specified amount *m* to the Consumer
takeMoney()	request from DVMController to release all the money currently inserted into the first collector to the second collector
releaseMoney()	request from DVMController to eject all the money currently inserted into the first collector
PricePanel	panel used by Service Person to set for each shelf the price of its drinks
setPriceOfShelf(s:Shelf,price:Money)	the price of drinks in shelf *s* was set to *price*
InformationPanel	component to display status information; different kinds of information can be displayed (e.g. text, graphics, lights, etc)
displayMoney(m:Money)	request from DVMController to display *m* as the amount of money currently inserted into the first collector

insufficientFunds(on:Boolean)
: if *on=true*: request from DVMController to display a text saying that the money currently inserted into the first collector is not sufficient to pay for the selected drink, if *on=false*: request from DVMController to erase an earlier text of this kind;

drinkNotAvailable(on:Boolean)
: similar to insufficientFunds; the shelf of the selected drink is empty

outOfService(on:Boolean)
: similar to insufficientFunds; the system is out of order and waits for maintenance by a Service Person

5 Concept Model

Once the messages are identified, the next goal is to describe how the system behaves after receiving and dispatching input messages. However, the behavior depends on internal states of the DVMController and thus we need a description of the internal structure of the DVMController first. The Concept Model in Fig. 4 provides such a description by a class diagram.

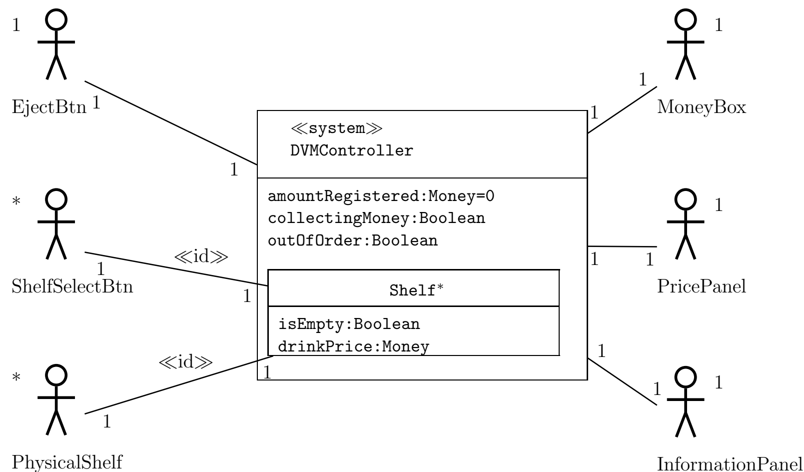

Fig. 4. Concept Model of DVMController

The attribute *amountRegistered* enables the *DVMController* to keep track of the money currently inserted into the first collector. The attributes *collectingMoney* and *outOfOrder* record the current state of the *DVMController* (see also Protocol Model in Section 6).

The component class *Shelf* is a logical representation of the component *PhysicalShelf* and has two attributes with obvious meanings. The two associations between *PhysicalShelf*, *ShelfSelectBtn*, and *Shelf* and the chosen

multiplicities ensure that every *ShelfSelectBtn* belongs to exactly one *PhysicalShelf*. The ≪id≫ stereotype means that the system can identify an actor starting from an object belonging to the system, e.g., given a shelf *s*, we can find its corresponding physical shelf, denoted in OCL by *s.physicalShelf*. The reason for the ≪id≫ stereotyped association is that the system can only send a message to an actor that can be identified. Identifying an external actor form inside the system will be the only use of ≪id≫ stereotyped associations.

6 Protocol Model

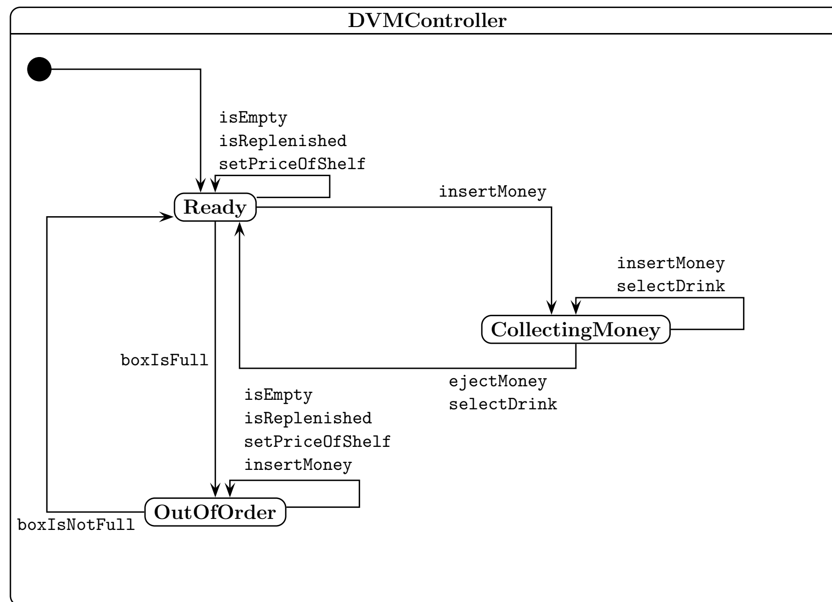

Fig. 5. Protocol Model of DVMController

The Protocol Model defines the temporal ordering of system operations. Each input message for the DVM from the environment refers to an *asynchronous operation invocation* and the corresponding event. The message, the event, and the invoked system operation have usually the same name.

A Protocol Model is described with a UML statechart which has no guards. As usual, a transition in the Protocol Model is triggered by an input event only if the system is in a state to dispatch it, i.e., there exists an arc with the name of the input event. If not, the input event is ignored. A transition from one state to another leads to the execution of the system operation with the same name as the input event.

We only give a brief informal explanation of the intended meaning of the states in Fig. 5. The state *Ready* represents the situation where both the Consumer and the Service Person are allowed to interact with the system. There is no money in the first collector, and no status information are displayed on the information panel.

The state *Ready* is left once the event *insertMoney* is dispatched. The target state is *CollectingMoney*. Further coins can be inserted into the first collector due to the self-transition from *CollectingMoney* to *CollectingMoney* triggered by event *insertMoney*. Note that once it is full the first collector will reject additional coins automatically without sending an *insertMoney* message to the DVMController. This way, the capacity problem of the first collector is solved “mechanically” by the actor *MoneyBox* itself.

Dispatching the event *ejectMoney* in state *CollectingMoney* causes a transition to *Ready*. After dispatching the event *selectDrink* in state *CollectingMoney* the triggered transition is nondeterministic. The controller can change its state to *Ready* or can remain in the state *CollectingMoney*, depending on the amount of inserted money and the availability of drinks. Nondeterministic state changes are permitted in Protocol Models because the nondeterminism usually disappears once the operation behavior is taken into account.

In state *Ready* the events *selectDrink* and *ejectMoney* are ignored. This is justified because in state *Ready* there is no money in the first collector neither to pay for a selected drink nor to give it back to the Consumer. Note, that the situation in state *OutOfOrder* is different for event *insertMoney*. Although, the DVM waits for maintenance by a Service Person when being in state *OutOfOrder*, the event *insertMoney* cannot be ignored here. *InsertMoney* belongs to the group of non-ignorable events due to the requirements. Recall that *insertMoney* occurs after the Consumer has successfully introduced a coin into the first collector, i.e. the Consumer has already paid and expects to get a drink or at least to get the money back. Note, that for preventing the Consumer from inserting a coin when the machine is out of order, we would need further assumptions, e.g. that the first collector gets locked.

In state *Ready* the event *isEmpty* as well as *boxIsFull* can be dispatched causing a state transition to *Ready* and *OutOfOrder*, respectively. The two events *isReplenished* and *setPriceOfShelf* cause a self-transition when dispatched in state *Ready* or *OutOfOrder*. The event *boxIsNotFull* is ignored in state *Ready*. It causes a state transition to *Ready* when dispatched in state *OutOfOrder*.

6.1 Synchronization Problems

Assuming the standard semantics for UML statecharts the Protocol Model presented above is still incorrect. An additional assumption on synchronized message handling is needed to solve the following problem:

The messages *isEmpty* and *boxIsFull* are sent once it is detected that a shelf is empty or the money box (second collector) has exceeded its capacity. For the correct working of the DVM it is necessary to assume that all other messages which are sent later are also processed later. However, this cannot be guaranteed if the current semantics of UML statecharts is assumed.

Suppose, besides *isEmpty* and *boxIsFull* there is another event, say *in-*

sertMoney, in the input queue of the DVM. It would be possible to dispatch *insertMoney* before *isEmpty* and *boxIsFull* although it was raised after them. Once the DVM has processed *insertMoney* it is in state *CollectingMoney* in which *isEmpty* and *boxIsFull* would be simply ignored. This could obviously lead to incorrect behavior.

The described effect is an inherent problem of message based asynchronous systems.

7 Operation Model

Not all aspects of the system behavior are expressed in the Protocol Model so far, e.g. we would like to specify that by inserting money into the first collector the money display is updated as well. Fortunately, the missing information can be expressed in an elegant form by OCL constraints.

The Operation Model contains detailed behavioral specification of the effect of all input messages. Since the input messages are solely operation calls, there is one corresponding operation for each input message usually with the same name.

The specification is mainly given by OCL pre-/postconditions. For the sake of brevity, a generalized *frame assumption* is taken as granted for the semantics of the postconditions. The frame of the specification is the list of all variables that can be changed by the operation [6]. 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. One approach that avoids this extra work is to imply a "... and nothing else changes" rule when dealing with specifications [1]. 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.

Most of the operations produce output messages which are sent to actors. Thus, the specification takes advantage of the message construct recently incorporated into OCL. The expression `receiver^message` can only occur in the postcondition of an operation and indicates that during the execution of the operation the message `message` has been sent to the object `receiver`. For a detailed account see [8, Section 2.7].

Another peculiarity of our specification is the usage of the keyword *sender*. This keyword is not part of the OCL standard but has proven to be extremely useful in our specification. The keyword *sender* refers to that actor which has invoked the operation. Recall, that every operation in Fondue is only invoked after a message with the same name was dispatched. The keyword *sender* allows an access to the sender of the dispatched message within OCL

expressions. It is similar to the keyword *self* which provides access to the object on which the operation has been invoked.

Operation: DVM::ejectMoney()

```
Use Cases: buy drink
Messages: InformationPanel::{DisplayMoney, InsufficientFunds,
                    DrinkNotAvailable}
          MoneyBox::{ReleaseMoney}
Pre :
Post :    self.amountRegistered = 0 and
          self.moneyBox^releaseMoney() and
          self.informationPanel^displayMoney(0) and
          self.informationPanel^insufficientFunds(false) and
          self.informationPanel^drinkNotAvailable(false) and
          self.collectingMoney = false
```

Operation: DVM::selectDrink()

```
Use Cases: buy drink
Messages: InformationPanel::{DisplayMoney, InsufficientFunds,
                    DrinkNotAvailable}
          MoneyBox::{ReturnMoney, TakeMoney}
          PhysicalShelf::{GiveDrink}
Pre :
Post :    if sender.shelf.isEmpty then
                    self.informationPanel^drinkNotAvailable(true) and
                    self.informationPanel^insufficientFunds(false) and
                    self.collectingMoney=true
          elsif self.amountRegistered < sender.shelf.drinkPrice then
                    self.informationPanel^insufficientFunds(true) and
                    self.informationPanel^drinkNotAvailable(false) and
                    self.collectingMoney=true
          else – successful case, Consumer gets drink
                    self.amountRegistered = 0 and
                    self.moneyBox^takeMoney() and
                    self.moneyBox^returnMoney(self.amountRegistered@pre
                    - sender.shelf.drinkPrice) and
                    sender.shelf.physicalShelf^giveDrink() and
                    self.informationPanel^displayMoney(0) and
                    self.informationPanel^insufficientFunds(false) and
                    self.informationPanel^drinkNotAvailable(false) and
                    self.collectingMoney=false
```

Operation: DVM::isEmpty()

```
Use Cases: buy drink
Pre :
Post :    sender.shelf.isEmpty = true
```

Operation: DVM::isReplenished()
Use Cases: service DVM
Pre :
Post : sender.shelf.isEmpty = false

Operation: DVM::insertMoney(m:Money)
Use Cases: buy drink
Messages: InformationPanel::{DisplayMoney, InsufficientFunds, DrinkNotAvailable}
MoneyBox::{ReleaseMoney}
Pre :
Post :
```
if not(self.outOfOrder) then
        self.amountRegistered = self.amountRegistered@pre + m and
        self.informationPanel^displayMoney(self.amountRegistered) and
        self.informationPanel^insufficientFunds(false) and
        self.informationPanel^drinkNotAvailable(false) and
        self.collectingMoney = true
else
        self.moneyBox^releaseMoney()
```

Operation: DVM::boxIsFull()
Use Cases: buy drink, service DVM
Messages: InformationPanel::{OutOfService}
Pre :
Post :
```
self.informationPanel^outOfService(true) and
self.outOfOrder = true
```

Operation: DVM::boxIsNotFull()
Use Cases: service DVM
Messages: InformationPanel::{OutOfService}
Pre :
Post :
```
self.informationPanel^outOfService(false) and
self.outOfOrder = false
```

Operation: DVM::setPriceOfShelf(s:Shelf,price:Money)
Use Cases: service DVM
Pre :
Post : s.drinkPrice = price

In the postcondition of operation *selectDrink()* the attribute *collectingMoney* is set in order to indicate whether the DVM remains in state *CollectingMoney* or takes the transition to state *Ready*. This resolves the non-determinism in the Protocol Model mentioned in Section 6. Unfortunately, the specification has to keep track of the value of *collectingMoney* in all operations which can enter or exit state *CollectingMoney*, i.e. the postcondition of *insertMoney* and *ejectMoney* has to set *collectingMoney* as well. This is rather clumsy since the post-state value is already known from the Protocol Model.

It would be possible to get rid of attribute *collectingMoney* and to use in the postcondition the construct *oclInState* (e.g. *self.oclInState(CollectingMoney)* instead of *self.collectingMoney = true*).

We did not apply this possibility yet, since it also introduces new dependencies between the Operation Model and Protocol Model whose consequences for the Fondue method have not yet been assessed.

8 Conclusion and Lessons Learnt

In this case study we gained some insights on the usage of OCL in the analysis phase of software development. Some of the lessons learnt might be qualified as Fondue-specific, but most of them are applicable in other cases as well.

simple statecharts Fondue restricts the Protocol Model to a simple kind of statecharts. As seen in the case study, a lot of information usually captured in statecharts (e.g. guards, raised actions) can also be expressed as pre-/postconditions. The raising of actions can now be expressed by OCL, because of the new message construct introduced into OCL 2.0.

oclInState If the OCL specification needs information about the current Protocol Model state, there are basically two possibilities. One can either introduce a new attribute encoding the state, or one can use the construct *oclInState*, which was incorporated into OCL recently (for a discussion of its semantics the reader is referred to [4]). We discussed both solutions in our case study and pointed out their advantages and disadvantages.

sender In the analysis phase of Fondue, operations are always *system operations*, which can be invoked only after a corresponding input event has been dispatched. Sometimes, we have to know the sender of the message, for instance when an internal data structure (e.g. *shelf*) representing the state of the sending actor (e.g. *PhysicalShelf*) has to be updated (e.g. *sender.shelf.isEmpty = true*).

Like the keyword *self* the keyword *sender* represents in Fondue a part of the "context" of the current operation.

As we have seen, the solely use of asynchronous messages for modeling the interaction between a system and outside actors can lead to synchronization problems. We plan to investigate the possibilities of using synchronous blocking calls, but also communication mechanisms based on memory shared between the system and an actor.

We are aware that the shown specification might still contain errors and inconsistencies because the specification is only paperwork so far. The next step in our research will be the development of a toolset in order to animate of effects of Fondue specifications.

The "end user" can then check by observation that the specification meets his expectations, and the specifier what he wrote is consistent with what he thought.

References

[1] A. Borgida, J. Mylopolous, and R. Reiter. ...And Nothing Else Changes: The Frame Problem in Procedure Specifications. In *Proceedings of ICSE-15*, pages 303–314. IEEE Computer Society Press, 1993.

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

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

[4] S. Flake and W. Müller. Semantics of State-Oriented Expressions in the Object Constraint Language. In *Proceedings of 15th International Conference on Software Engineering and Knowledge Engineering (SEKE 2003)*, pages 142–149. Knowledge Systems Institute, 2003.

[5] C. B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

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

[7] OMG. *OMG Unified Modelling Language Specification, Version 1.5*, Mar. 2003. Available at: `www.omg.org/technology/documents/formal/uml.htm`.

[8] Response to the UML 2.0 OCL RfP, Version 1.6. OMG Document ad/2003-01-07, Jan 2003.

[9] S. Sendall. *Specifying Reactive System Behavior.* PhD thesis, Swiss Federal Institute of Technology in Lausanne, School of Computer and Communication Sciences, 2002. No 2588.

[10] S. Sendall and A. Strohmeier. UML Based Fusion Analysis Applied to a Bank Case Study. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 278–291. Springer, 1999. An extended version also available as Technical Report EPFL-DI No 99/319.

[11] S. Sendall and A. Strohmeier. From Use Cases to System Operation Specifications. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 1–15. Springer, 2000. Also available as Technical Report EPFL-DI No 00/333.

[12] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1992.