DOI 10.1007/s00165-007-0062-3
BCS © 2007
Formal Aspects of Computing (2008) 20: 79–100

Formal Aspects of Computing

Modeling and validating Mondex scenarios described in UML and OCL with USE

Mirco Kuhlmann and Martin Gogolla

Computer Science Department, Database Systems Group, University of Bremen, 28334 Bremen, Germany.
E-mail: opti@informatik.uni-bremen.de; gogolla@informatik.uni-bremen.de

Abstract. This paper describes the Mondex case study with UML class diagrams and restricting OCL constraints. The constraints have been formulated either as OCL class invariants or as OCL pre- and postconditions. The proposed two models include UML class diagrams and OCL constraints which have been checked by the UML and OCL tool USE (UML-based Specification Environment). USE allows validation of a model by testing it with scenarios. The Mondex case study has been validated by positive and negative test cases. The test cases allow the validity of the various constraints to be traced and checked. Validation results are presented as textual protocols or as UML sequence diagrams where starting, intermediate, and resulting system states are represented by UML object diagrams. UML sequence diagrams, UML object diagrams, and textual protocols are shown with varying degrees of detail for the attributes, constraints, and executed commands.

1. Introduction

This paper is a UML and OCL [RJB05, Gro05, WK03] description of central aspects of the Mondex case study originally proposed in [SCW00]. The abstract of that paper reads as follows:

This case study is a reduced version of a real development by the NatWest Development Team of a Smartcard product for electronic commerce. This development was deeply security critical: it was vital to ensure that these cards would not contain any bugs in implementation or design that would allow them to be subverted once in the field. The system consists of a number of electronic purses that carry financial value, each hosted on a Smartcard. The purses interact with each other via a communications device to exchange value. Once released into the field, each purse is on its own: it has to ensure the security of all its transactions without recourse to a central controller. All security measures have to be implemented on the card, with no realtime external audit logging or monitoring. We develop two key models in this case study. The first is an abstract model, describing the world of purses and the exchange of value through atomic transactions, expressing the security properties that the cards must preserve. The second is a concrete model, reflecting the design of the purses which exchange value using a message protocol. Both models are described in the Z notation, and we prove that the concrete model is a refinement of the abstract.

In this work we only considered the abstract Mondex model, not the concrete model and consequently no refinement proofs. We have concentrated on the abstract model, because it is the one which could be discussed with a client in order to demonstrate which aspects of the system will be handled and which system properties will be maintained. A client should understand the central formal requirements, at least in rudimentary form. A further reason for not considering the concrete model and the refinement proofs was resource limitation. However, both the concrete model and the refinement could be handled with UML and OCL, although a formal refinement notion is not available in OCL yet.

We have treated the case study with the OCL validation system USE [RG01, GBR05]. Roughly speaking, USE is an interpreter for a subset of UML and full OCL: USE covers UML class diagrams with invariants and

Correspondence and offprint requests to: M. Kuhlmann, E-mail: opti@informatik.uni-bremen.de pre- and postconditions and UML object and sequence diagrams. Positive and negative test scenarios can be realized in USE with command sequences and can be visualized with UML sequence diagrams. Failure detection and inspection for invariants and pre- and postconditions can be realized with the so-called evaluation browser. System state inspection is done with object diagrams and OCL queries. Due to resource restrictions, we did not use the power of the USE system state generator [GBR05] although one of our models which we have developed for the Mondex case study is well-suited for that generator.

The purpose for employing USE was to demonstrate central properties of the described abstract model in a user-friendly and easy-to-understand form, i.e., with graphical representations which in our case means UML class, object, and sequence diagrams. The purpose of our activities was not to prove properties. Other systems and approaches like [But06, RJ06, Jon06, Woo06, Cro06, SGHR06] concentrated on the proof aspect and maybe stronger than USE for that. We think that a client and of course the developer should get a clear picture of the formal model and its implications because the formal model is the starting point for further development. Formal descriptions are hard to follow and our intention was to help in understanding such complex formal descriptions.

We have handled the Mondex case study in two styles: we have developed what we call an imperative approach and a declarative approach. The imperative approach realizes the temporal development of a purse collection with ordinary operations changing the status of a single purse in each step and checking before, respectively, after each step whether the system invariants and pre-, respectively, postconditions of the operations are respected. The declarative approach records the temporal development of a purse collection in a single structure in which the single steps of the imperative approach are encoded and checks the constraints with boolean expressions determining the validity of the structure. We systematically and symmetrically handle both approaches insofar that the same scenarios, i.e., one mainstream scenario and different exceptional scenarios, are treated in both approaches.[1] UML and OCL do not force one to use one approach, the imperative one or the declarative one, or favor one approach over the other. Our modeling of money in form of coins and the declarative approach are quite similar to the way the Mondex case study is handled in [RJ06]. Similar results as ours could be achieved with tools like the OCL engine of ATL [CPC$^+$04] or the OCLE system [JAB$^+$06].

The structure of the rest of the paper is as follows. Section 2.1 introduces the class diagram for the imperative approach, whereas Sect. 2.2 explains the class diagram for the declarative one. Section 3.1 discusses the mainstream scenario in the imperative approach and Sects. 3.2–3.4 handle the exceptional scenarios. Section 4.1 deals with the mainstream scenario in the declarative approach and Sects. 4.2–4.4 show the exceptional scenarios in the declarative approach. Section 5 compares both approaches and points out advantages and disadvantages. Section 6 finishes the paper with some concluding remarks. To get a quick overview on the paper, the Sects. 2.2 and 4 treating the declarative approach may be skipped. The reader will however miss then the query and evaluation browser facilities of the approach.

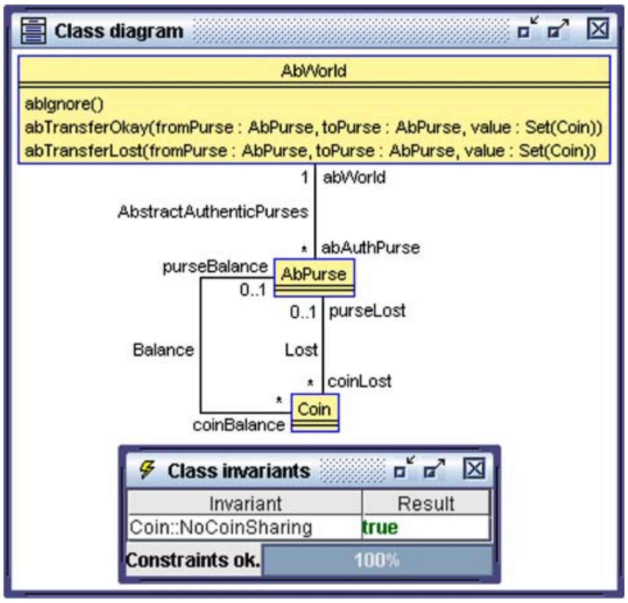
2. Class diagrams

The Mondex case study specifies an abstract model defining the world of abstract purses and the elements relevant to system security. They are described here with two different UML models following what we call an imperative and a declarative approach. Both realize abstract worlds, purses and their balance and lost values which are represented by sets of coins. The imperative approach uses the Object Constraint Language for modeling the abstract operations $AbIgnore$, $AbTransferOkay$ and $AbTransferLost$. Pre- and postconditions define the corresponding security properties to ensure correct operation calls and executions. The USE system provides the possibility to animate the execution of operations without or with side effects. This feature is used to show the system evolution in the following sections.

The declarative approach specifies the abstract operations as boolean query operations which are side effect free. These OCL operations access the security properties via predicates likewise defined as boolean queries. The system evolution is explicitly modeled as a chain of abstract worlds wherein a transition consisting of one world and its direct successor implicitly represents the execution of one of the abstract (boolean) operations $AbIgnore$, $AbTransferOkay$ or $AbTransferLost$. A validation instance checks if the transitions correspond to correct operation executions, i.e., for every pair of successive worlds one of the boolean operations should return *true*.

Compared to the model in [SCW00], there is no explicit finalization to observe the world elements. OCL invariants as well as pre- and postconditions can directly access the values defined within the available object

[1] All sources are available at [KG08].



Invariant	Result
Coin::NoCoinSharing	true

Constraints ok. 100%

Fig. 1. Class diagram—imperative approach

diagram. The preconditions present in the imperative approach read the values before the corresponding operation changes the state and the postconditions check them after the changes took place. In contrast to this, the validation object defined in the declarative approach is connected with the initial world of purses and navigates through the entire object diagram observing the components.

2.1. Imperative approach

The imperative approach specifies three UML classes which are connected via associations (see Fig. 1). The class `AbWorld` models abstract worlds. Every world has an arbitrary number of purses, modeled by the class `AbPurse`. It is not necessary to identify the purses with unique names because every purse has an object identity. The third class `Coin` represents the coins belonging to purses. The associations `Balance` and `Lost` connect this class with `AbPurse`. Thus a coin may be an element of the purse balance or the purse's set of lost coins. Coins model the minimum unit of money. The size of the sets of coins describing the purse balance (respectively, lost) values correspond to the natural numbers representing the values within the original Mondex model. In addition to the explicit description of money and the possibility to inspect the actions during an abstract transfer more precisely, further security properties may be specified. The later mentioned constraint `NoCoinSharing` does not allow the sharing of coins between different purses.

The abstract operations defined in the Mondex case study are present in class `AbWorld`. They are specified with OCL constraints in the form of pre- and postconditions which are discussed below. This study does not consider refinement proofs as in [SCW00, Sect. 3.3.1]. For this reason we do not define particular input and output types. Furthermore it is not required that operations return dummy values. The association `Abstract-AuthenticPurses` establishes a connection between an abstract world and authentic purses. Its multiplicities require that a purse must belong to exactly one world. It is possible to navigate from purse objects to their world via the role name `abWorld`. The navigation from an abstract world object via the role `abAuthPurse` results in the set of authentic purses.

Purses do not have attributes. Their balance and lost components are modeled with the associations `Lost` and `Balance` as mentioned above. The set of coins reached with the role name `coinBalance` (respectively, `coinLost`) represent the purse's balance (respectively, the purse's lost) value. Both sets may contain an arbitrary number of coins, but every coin must belong to exactly one purse. In addition they must be connected either with a `Balance` link or a `Lost` link to the purse. The multiplicities 0..1 (at `purseBalance` and `purseLost`) constrain the number of links, but they are not sufficient to ensure the postulated properties. The OCL invariant `NoCoinSharing` fills this gap with the expression `purseBalance<>oclUndefined(AbPurse) xor purseLost<>oclUndefined(AbPurse)`.

The pre- and postconditions of the three abstract operations constrain the states before the operations are called and the states after operation termination. The abstract operation `abIgnore` has no formal parameters and no preconditions are specified. `abIgnore` does not change the world. This fact is described with one postcondition.

```
abIgnore()
  post WorldDoesNotChange :
    abAuthPurse = abAuthPurse@pre and
    abAuthPurse@pre->forAll( purse | purse.coinBalance = purse.coinBalance@pre and
                                     purse.coinLost = purse.coinLost@pre )
```

The postcondition name is `WorldDoesNotChange`. It is defined with a conjunction of two requirements. Its first operand states, that the set of authentic purses belonging to the world in the current state (`abAuthPurse`) must equal the set in the previous state (`abAuthPurse@pre`). The postfix `@pre` refers to the state before the operation was invoked. It allows to access previous values. The second operand quantifies over all authentic purses. All balance and lost values in the current state must equal their previous values. A requirement like `abAuthPurse = abAuthPurse@pre` could even be extended in USE by demanding `AbPurse.allInstances = AbPurse.allInstances@pre`. However, the `@pre` applied to a class is not part of standard OCL (whereas `allInstances` is in the standard and returns the current set of objects of the respective class).

The operations `abTransferOkay` and `abTransferLost` have three formal parameters specifying the *from* and *to* purses and the coins which should be transferred. It is allowed to transfer an empty set of coins. The parameters represent the transfer details which are explicitly modeled in the declarative approach following below. Both operations are constrained with the same preconditions. The *from* and *to* purses must be included in the world's set of authentic purses and must differ. Furthermore the *from* purse has to own the coins to be transferred before the operation is invoked. Three postconditions require that the set of abstract purses and the balance and lost values of purses not participating in the transfer do not change and the balance of the *from* purse decreases in the set of transferred coins. They exist in both operations.

A correct execution of `abTransferOkay` results in an increased balance of the *to* purse which is claimed by the postcondition `ToPurseBalanceIncreases`.

```
post ToPurseBalanceIncreases :
  toPurse.coinBalance = toPurse.coinBalance@pre->union( value )
```

The set of coins in the current state is supposed to contain the same coins as the union of the previous set and the transferred coins indicated by `value`. Two further postconditions require that the lost values of the involved purses do not change. The constraints of `abTransferLost` are defined analogously. There the postcondition `FromPurseLostIncreases` replaces `ToPurseBalanceIncreases` and a further constraint does not allow the *to* purse's balance and lost values to change.

Beside the aforementioned constraints, we defined the security properties `NoValueCreation` and `AllValue-Accounted` in OCL. They are stated and proved in [SCW00].

```
post NoValueCreation :
  abAuthPurse@pre->collect( p | p.coinBalance@pre )->flatten()->includesAll(
    abAuthPurse->collect( p | p.coinBalance )->flatten() )

post AllValueAccounted :
  abAuthPurse->collect( p | p.coinBalance )->flatten()->union(
    abAuthPurse->collect( p | p.coinLost )->flatten() ) =
  abAuthPurse@pre->collect( p | p.coinBalance@pre )->flatten()->union(
    abAuthPurse@pre->collect( p | p.coinLost@pre )->flatten() )
```

Originally the postcondition `NoValueCreation` requires the sum of all the purses' balances not to be increased. Consequently all coins existing as balance values after the execution of an abstract operation must be a member of the corresponding set available before the operation was invoked. Both sets are not forced to be equal, because coins may get lost. We calculate the desired sets by collecting the balance values of every purse, which are in each case represented as a set of coins, and by flattening the result afterwards.[2] In addition to this requirement, the

[2] In fact the OCL collection operation **collect** results in a bag. But this situation has no influence on the result, because all included sets are disjoint.

postcondition AllValueAccounted specifies, that the set of all coins, i.e., the union of all balance and lost coins existing in an abstract world, is not allowed to change during the evolution.

2.2. Declarative approach

The declarative approach in Fig. 2 is designed to model the evolution of an abstract world explicitly. For this purpose, the reflexive association Order is defined. It allows the creation of abstract world chains beginning in an initial world (AbInitWorld) and ending in an end world (AbEndWorld). Both are subclasses of AbWorld. The direct successor or predecessor of an abstract world is reached via the role name succ or pred. As in the imperative approach, every world includes a set of abstract purses which are connected with coins via the associations Balance and Lost. This set of purses must not change in a valid world chain. This implicates, that a purse has to exist in different worlds. But there is only one balance and one lost value defined for a single purse object. Their values may change during a valid transfer, i.e., a transition from a world to its successor. According to that, one abstract purse must be represented by different objects which have a name identifying the purse. Having a valid world chain implies that every world is connected with its own purse copies. These aspects will be again discussed when we consider the object diagram in Fig. 10.

The class WorldsValidation and its connection with AbInitWorld plays a crucial role in terms of system security. A world chain is considered as valid if the boolean operation validate returns *true*. In this case every transition from one world to the next must correspond to a correct abstract transfer. The auxiliary operation validateWorldChain is invoked by validate and checks the world chain recursively. It passes all pairs of directly linked world objects to the operation abTransfer. This query operation is accessible in the connected class AbOperations and is defined as the disjunction of the boolean OCL queries abIgnore, abTransferOkay and abTransferLost which also require two objects of successive worlds as actual parameters. The predecessor world is connected to an instance of the class TransferDetails describing the action performed to reach the successor world. It includes the *from* and *to* purse and the transferred value. The attribute values of TransferDetails objects equal oclUndefined if an abstract ignore is performed. If the transfer details together with the changes characterized by the pair of worlds fit to one abstract operation, the corresponding OCL query returns *true*.

The abstract operations defined within the imperative specification change the system state. The query operations described in the declarative approach cannot change it. That is why the class MetaOperations is defined. It includes meta operations for creating the world chain. The first operation to be invoked is createInitialWorld. It creates a specific number of coins, the abstract purses, the initial world and the validation instances. The other operations create a new world representing the resulting world after an abstract ignore or okay (respectively, lost) transfer was performed. It is linked to the predecessor world created during the last meta operation call. Furthermore every meta operation except createInitialWorld creates an instance of the transfer details and connects it with the predecessor. The arguments of createOkayWorld and createLostWorld specify the values of the *from* and *to* purse and the set of transferred coins.

To establish an analogy to the imperative approach, the security properties in the declarative approach are explicitly defined as boolean queries named after the corresponding pre- and postconditions (see Fig. 3). The security properties are split into four classes (CommonConstraints, IgnoreConstraints, OkayConstraints, LostConstraints) to make clear in which abstract operation the boolean operations are used as security predicates. Both abTransferOkay and abTransferLost check the properties defined in the class CommonConstraints. But the operations specified in class OkayConstraints (respectively, LostConstraints) exclusively belong to abTransferOkay (respectively, abTransferLost). The query worldDoesNotChange of class IgnoreConstraints is only used in abIgnore.

The definition of the security properties have to be adapted to the fact, that predecessor and successor worlds are modeled explicitly and the purses are identified by names. The implicitly accessible pair of system states in pre- and postconditions and the postfix @pre are not available in this approach. The realization of the constraints considered in Sect. 2.1 is shown below. All query operations shown in Fig. 3 need a pair of abstract worlds to check whether the security property they represent holds.

```
worldDoesNotChange( predWorld : AbWorld, succWorld : AbWorld ) : Boolean =
  predWorld.succ = succWorld and
  predWorld.abAuthPurse->size() = succWorld.abAuthPurse->size() and
  succWorld.abAuthPurse->forAll ( postPurse |
    predWorld.abAuthPurse->exists ( prePurse |
```

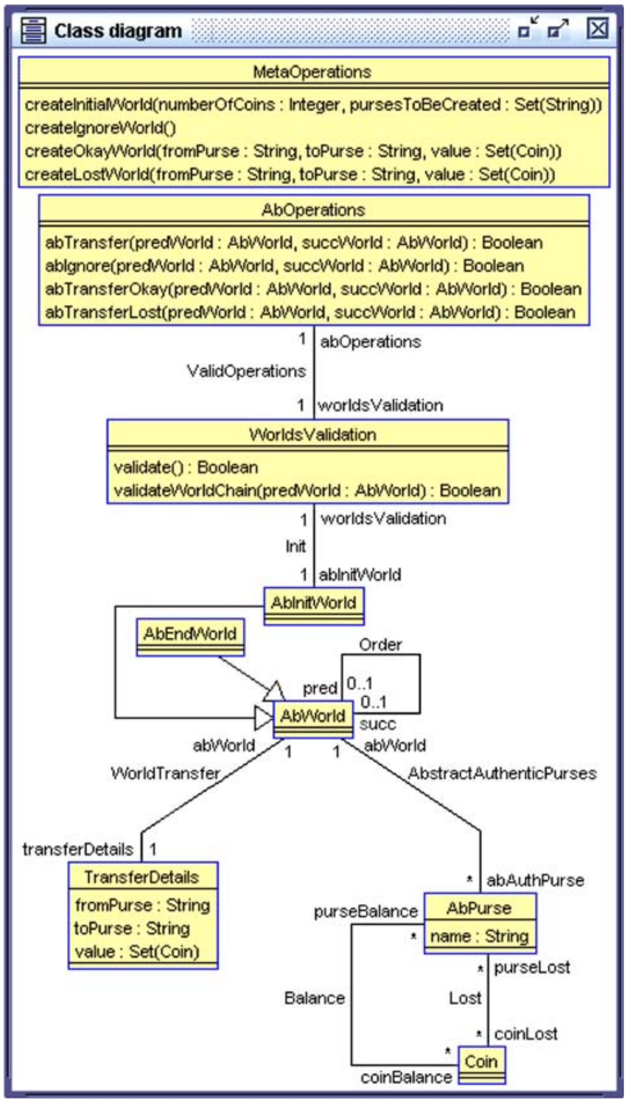
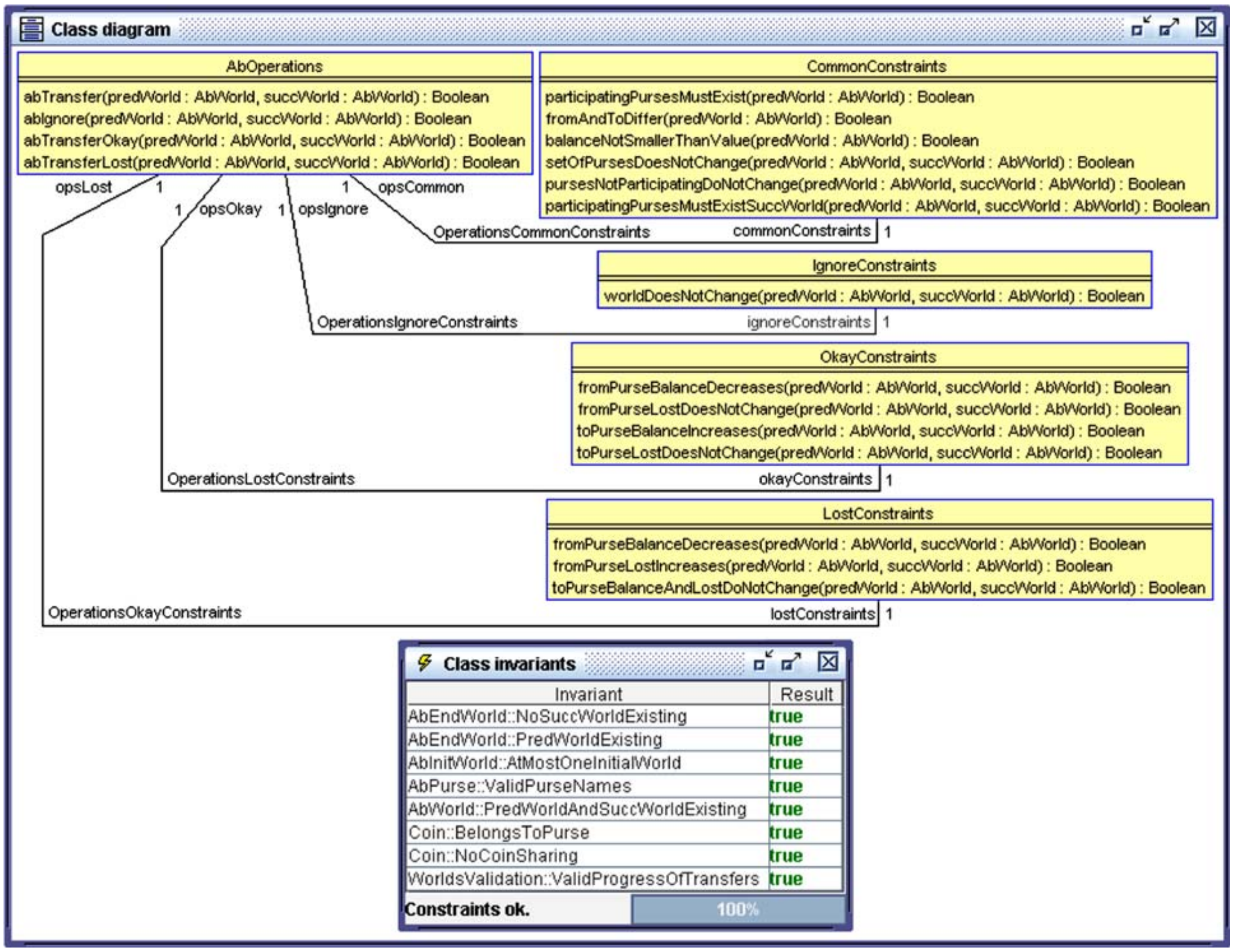

Class diagram
MetaOperations
createInitialWorld(numberOfCoins : Integer, pursesToBeCreated : Set(String))
createIgnoreWorld()
createOkayWorld(fromPurse : String, toPurse : String, value : Set(Coin))
createLostWorld(fromPurse : String, toPurse : String, value : Set(Coin))
AbOperations
abTransfer(predWorld : AbWorld, succWorld : AbWorld) : Boolean
abIgnore(predWorld : AbWorld, succWorld : AbWorld) : Boolean
abTransferOkay(predWorld : AbWorld, succWorld : AbWorld) : Boolean
abTransferLost(predWorld : AbWorld, succWorld : AbWorld) : Boolean
1 abOperations
ValidOperations
1 worldsValidation
WorldsValidation
validate() : Boolean
validateWorldChain(predWorld : AbWorld) : Boolean
1 worldsValidation
Init
1 abInitWorld
AbInitWorld
AbEndWorld
Order
pred 0..1
0..1
succ
AbWorld
abWorld 1
1 abWorld
WorldTransfer
AbstractAuthenticPurses
transferDetails 1
TransferDetails
fromPurse : String
toPurse : String
value : Set(Coin)
* abAuthPurse
purseBalance
AbPurse
* name : String
* purseLost
Balance
Lost
* coinLost
*
Coin
coinBalance

Fig. 2. Class diagram—declarative approach

```
      postPurse.name = prePurse.name ) ) and
  predWorld.abAuthPurse->forAll ( prePurse |
    succWorld.abAuthPurse->exists ( postPurse |
      prePurse.name = postPurse.name ) ) and
  predWorld.abAuthPurse->forAll( pPre |
      succWorld.abAuthPurse->forAll( pPost |
        pPre.name = pPost.name implies
          ( pPre.coinBalance = pPost.coinBalance and
            pPre.coinLost = pPost.coinLost ) ) )
```

This operation returns *true* if and only if the arguments represent successive worlds, the sets of abstract purses belonging to the worlds have the same size, for every purse object in the successor world there exists a purse object in the predecessor world with the same name, for every purse object in the predecessor world there exists a purse object in the successor world with the same name, and the pair of purse objects in the predecessor and successor world referring to the same purse have the same balance and lost values.


Class diagram
AbOperations
abTransfer(predWorld : AbWorld, succWorld : AbWorld) : Boolean
abIgnore(predWorld : AbWorld, succWorld : AbWorld) : Boolean
abTransferOkay(predWorld : AbWorld, succWorld : AbWorld) : Boolean
abTransferLost(predWorld : AbWorld, succWorld : AbWorld) : Boolean
CommonConstraints
participatingPursesMustExist(predWorld : AbWorld) : Boolean
fromAndToDiffer(predWorld : AbWorld) : Boolean
balanceNotSmallerThanValue(predWorld : AbWorld) : Boolean
setOfPursesDoesNotChange(predWorld : AbWorld, succWorld : AbWorld) : Boolean
pursesNotParticipatingDoNotChange(predWorld : AbWorld, succWorld : AbWorld) : Boolean
participatingPursesMustExistSuccWorld(predWorld : AbWorld, succWorld : AbWorld) : Boolean
opsLost 1
1 opsOkay
1 opsIgnore
1 opsCommon
OperationsCommonConstraints
commonConstraints 1
IgnoreConstraints
worldDoesNotChange(predWorld : AbWorld, succWorld : AbWorld) : Boolean
OperationsIgnoreConstraints
ignoreConstraints 1
OkayConstraints
fromPurseBalanceDecreases(predWorld : AbWorld, succWorld : AbWorld) : Boolean
fromPurseLostDoesNotChange(predWorld : AbWorld, succWorld : AbWorld) : Boolean
toPurseBalanceIncreases(predWorld : AbWorld, succWorld : AbWorld) : Boolean
toPurseLostDoesNotChange(predWorld : AbWorld, succWorld : AbWorld) : Boolean
OperationsLostConstraints
okayConstraints 1
LostConstraints
fromPurseBalanceDecreases(predWorld : AbWorld, succWorld : AbWorld) : Boolean
fromPurseLostIncreases(predWorld : AbWorld, succWorld : AbWorld) : Boolean
toPurseBalanceAndLostDoNotChange(predWorld : AbWorld, succWorld : AbWorld) : Boolean
OperationsOkayConstraints
lostConstraints 1
Class invariants

Invariant	Result
AbEndWorld::NoSuccWorldExisting	true
AbEndWorld::PredWorldExisting	true
AbInitWorld::AtMostOneInitialWorld	true
AbPurse::ValidPurseNames	true
AbWorld::PredWorldAndSuccWorldExisting	true
Coin::BelongsToPurse	true
Coin::NoCoinSharing	true
WorldsValidation::ValidProgressOfTransfers	true

Constraints ok. 100%

Fig. 3. Class diagram—declarative approach (constraints)

The next operation points out a further difference between the imperative and declarative approach. As mentioned above, the transfer details are not passed as arguments to the operations but are explicitly defined as a transfer details object. The expression `predWorld.transferDetails` refers to the navigation from the world object `predWorld` to an object of the class `TransferDetails` via the role name `transferDetails`. After reaching the connected object its attribute values can be accessed.

```
toPurseBalanceIncreases( predWorld : AbWorld, succWorld : AbWorld ) : Boolean =
  succWorld.abAuthPurse->forAll( postPurse |
    postPurse.name = predWorld.transferDetails.toPurse implies
      predWorld.abAuthPurse->forAll( prePurse |
        prePurse.name = predWorld.transferDetails.toPurse implies
          postPurse.coinBalance = prePurse.coinBalance->
            union( predWorld.transferDetails.value ) ) )
```

This operation searches for the purse object in the successor world representing the *to* purse specified in the transfer details. Its set of balance coins must equal the union of the previous set and the set of the transferred coins (`predWorld.transferDetails.value`). The previous set belongs to the purse object corresponding to the *to* purse in the predecessor world.

There are several invariants specified in order to constrain the world chain. `NoSuccWorldExisting` and `PredWorldExisting` are defined in the context of the class `AbEndWorld`. They state that an end world must not have a successor but a predecessor. In contrast to that, both predecessor and successor worlds have to be defined for

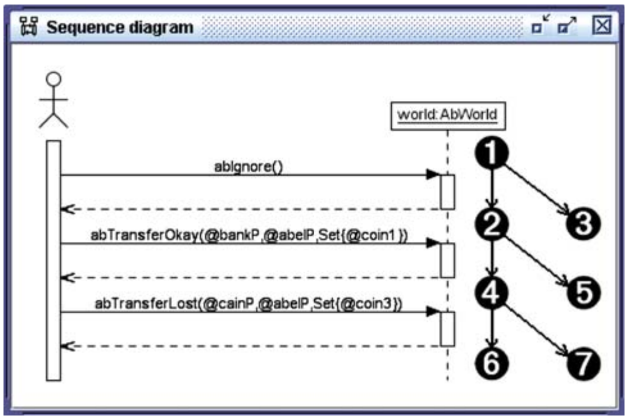

Sequence diagram
world:AbWorld
abIgnore()
abTransferOkay(@bankP,@abelP,Set{@coin1})
abTransferLost(@cainP,@abelP,Set{@coin3})
1
2
3
4
5
6
7

Fig. 4. Sequence diagram—imperative approach

objects of the super class `AbWorld` (`PredWorldAndSuccWorldExisting`). The invariant `AtMostOneInitialWorld` permits only one initial world existing in the system state, which implies that at most one world chain may be created. Furthermore all purse objects must have valid names comprising lower case letters, capitals and numbers (`ValidPurseNames`). There are two constraints defined in context of the class `Coin`.

```
inv BelongsToPurse :
  AbWorld.allInstances->forAll( world |
    world.abAuthPurse->exists( purse |
      ( purse.coinBalance->union( purse.coinLost ) )->includes( self ) ) )

inv NoCoinSharing :
  ( purseBalance->union( purseLost ) )->forAll( purse1, purse2 |
      purse1 <> purse2 implies
        purse1.abWorld <> purse2.abWorld )
```

The first invariant requires that every coin is linked to at least one purse object in every existing world. That means it is always an element of a balance or lost value. Unlike the corresponding invariant specified in the imperative approach, `NoCoinSharing` permits multiple links between a coin and different purse objects (from different worlds) in the declarative approach. That is because there is just one object created for every coin. The balance and lost values of the purse objects existing in the successive worlds continuously consist of the same coin objects. But coin sharing within one abstract world is still not allowed. If a coin belongs to two different purse objects (via `Lost` or `Balance` links), they must belong to two different worlds.

The last invariant invokes the operation `validate`. Its result consequently shows if the world chain describes a valid sequence of abstract transfers.

```
inv ValidProgressOfTransfers: validate()
```

The definition of `validate` and its auxiliary operation is presented below. The crucial part of the definition is the call of `abTransfer` for every pair of successive worlds existing in the world chain beginning in the initial world (`AbInitWorld`).

```
validate( ) : Boolean =
  abInitWorld <> oclUndefined( AbInitWorld ) and validateWorldChain( abInitWorld )

validateWorldChain( predWorld : AbWorld ) : Boolean =
  if predWorld.succ->isEmpty() then predWorld.oclIsTypeOf( AbEndWorld )
  else abOperations.abTransfer( predWorld, predWorld.succ ) and
       validateWorldChain( predWorld.succ ) endif
```

3. System evolution in the imperative approach

For analyzing the imperative and the declarative specification, different system states or object diagrams are created. They demonstrate various transfer scenarios which are realized with both models to clarify details of each model and the differences between the approaches. There is one mainstream scenario presenting an evolving world with four sequential states comprising three coins as well as three purses (abelP, bankP and cainP). The sequence diagram shown in Fig. 4 illustrates how the worlds evolve by invoking abstract operations defined as OCL operations with side effects within the imperative approach. The numbers indicate different object diagrams corresponding to one particular world. The object diagrams 1, 2, 4 and 6 describe the mainstream scenario whereas the object diagrams 3, 5 and 7 refer to exceptional scenarios where at least one security property is violated, i.e., at least one pre- or postcondition is false. Furthermore the sequence diagram lists the abstract operations leading from one system state to another. All abstract operations are considered, e.g., the operation `abIgnore` leading from the object diagram 1 to object diagram 2 in the mainstream scenario or to object diagram 3 in one exceptional scenario. The different scenarios are inspected in the following subsections.

3.1. Mainstream scenario

The mainstream scenario shown in Fig. 5 describes four object diagrams representing an abstract world evolving through three different operation calls which do not violate any security property. The initial world comprises the purses of Abel, which is linked to no coin, the bank having the coins `coin1` and `coin2` and Cain owning `coin3`. The links connecting the purses with their coins belong to the association `Balance`. That means no coin is lost yet. The invocation of `abIgnore` leads to object diagram 2 showing the same world because no changes took place. System state 4 is reached after calling `abTransferOkay` which transfers `coin1` from the bank's purse to Abel's purse. This transfer is indicated by the deletion of the `Balance` link between `coin1` and `bankP` and the creation of a new link between `coin1` and `abelP`. Nothing else changes. The last call describes a transfer where the coin to be transferred from Cain to Abel is lost. From this it follows that the balance link between `cainP` and `coin3` is replaced by a lost link in object diagram 6.

3.2. Exceptional scenario: world changes

Object diagram 3 results from an incorrect action performed within the operation `abIgnore` because it changes the world by creating a new purse instance and connecting it to the abstract world (see Fig. 6). This violates the postcondition `WorldDoesNotChange` defined for this operation. USE gives a message explaining the situation falsifying the constraint:

```
postcondition ‘WorldDoesNotChange’ is false
evaluation results:
  self : AbWorld = @world
  self.abAuthPurse : Set(AbPurse) = Set{@abelP,@bankP,@cainP,@newPurse}
  self : AbWorld = @world
  self.abAuthPurse@pre : Set(AbPurse) = Set{@abelP,@bankP,@cainP}
  (self.abAuthPurse = self.abAuthPurse@pre) : Boolean = false
  ((self.abAuthPurse = self.abAuthPurse@pre) and
    self.abAuthPurse@pre->forAll(purse : AbPurse |
      ((purse.coinBalance = purse.coinBalance@pre) and
        (purse.coinLost = purse.coinLost@pre)))) : Boolean = false
```

The expression `self.abAuthPurse = self.abAuthPurse@pre` with `self = @world` should be *true*, i.e., the set of authentic purses before and after the operation should be identical. But the set of purses in the pre state equals `Set{@abelP,@bankP,@cainP}` and the set after the termination of the operation equals `Set{@abelP,@bankP,@cainP,@newPurse}`.

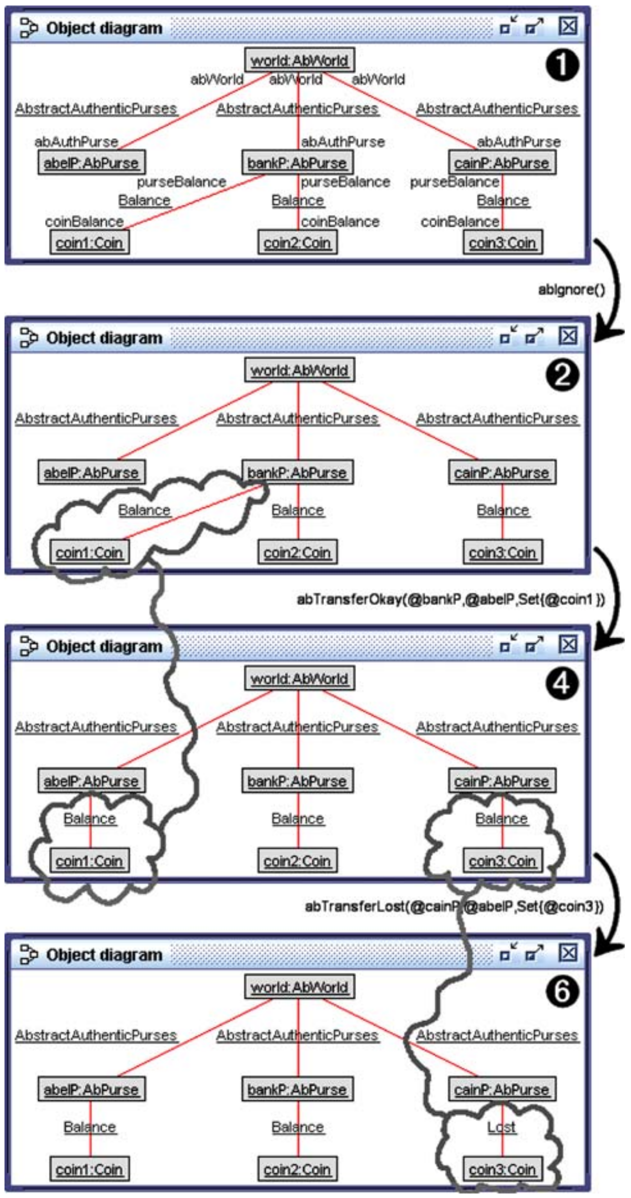


Fig. 5. Mainstream scenario of the imperative approach

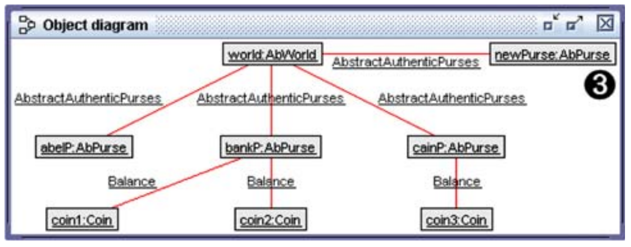


Fig. 6. AbIgnore—new purse created

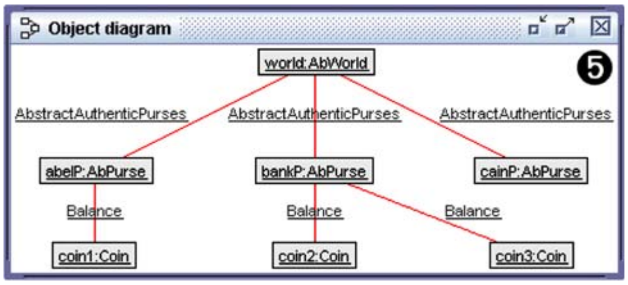


Fig. 7. AbTransferOkay—coin3 transferred to the Bank's purse

3.3. Exceptional scenario: further transfer

This scenario demonstrates an execution of `abTransferOkay` leading from object diagram 2 to object diagram 5. The latter diagram shows that beside `coin1` also `coin3` switches its owner (see Fig. 7). A secure transfer of balance forbids changes not indicated by the operation parameters. For that reason, `coin3` and Cain's purse should not be involved. Their participation violates the security properties `PursesNotParticipatingDoNotChange` and `FromPurseBalanceDecreases`. The USE system shows why this is the case.

```
postcondition 'PursesNotParticipatingDoNotChange' is false
evaluation results:
  self : AbWorld = @world
  self.abAuthPurse : Set(AbPurse) = Set{@abelP,@bankP,@cainP}
  fromPurse : AbPurse = @bankP
  toPurse : AbPurse = @abelP
  Set {fromPurse,toPurse} : Set(AbPurse) = Set{@abelP,@bankP}
  (self.abAuthPurse - Set {fromPurse,toPurse}) : Set(AbPurse) = Set{@cainP}
  purse : AbPurse = @cainP
  purse.coinBalance : Set(Coin) = Set{}
  purse : AbPurse = @cainP
  purse.coinBalance@pre : Set(Coin) = Set{@coin3}
  (purse.coinBalance = purse.coinBalance@pre) : Boolean = false
  ((purse.coinBalance = purse.coinBalance@pre) and
    (purse.coinLost = purse.coinLost@pre)) : Boolean = false
  (self.abAuthPurse - Set {fromPurse,toPurse})->forAll(purse : AbPurse |
    ((purse.coinBalance = purse.coinBalance@pre) and
      (purse.coinLost = purse.coinLost@pre))) : Boolean = false
```

The balance and lost value of all purses but the *from* and *to* purse must not change during the operation execution. This implies the truth of the expression `purse.coinBalance = purse.coinBalance@pre` with `purse` equal to `cainP`, i.e., the balance before the operation call and after its termination has to be the same. But this is not *true* in the exceptional scenario, because `coin3` is no longer existing in Cain's balance value.

```
postcondition 'FromPurseBalanceDecreases' is false
evaluation results:
  fromPurse : AbPurse = @bankP
  fromPurse.coinBalance : Set(Coin) = Set{@coin2,@coin3}
  fromPurse : AbPurse = @bankP
  fromPurse.coinBalance@pre : Set(Coin) = Set{@coin1,@coin2}
  value : Set(Coin) = Set{@coin1}
  (fromPurse.coinBalance@pre - value) : Set(Coin) = Set{@coin2}
  (fromPurse.coinBalance =
   (fromPurse.coinBalance@pre - value)) : Boolean = false
```

Furthermore the resulting balance of the bank's purse representing the *from* purse should be equal to the previous balance set without `coin1`, which is the value to be transferred. Formally, the value of the expression

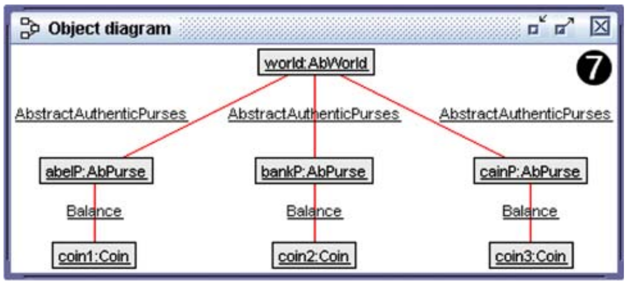


Fig. 8. AbTransferLost—no changes

`fromPurse.coinBalance@pre - value` must be equal to `Set{@coin2}` with `fromPurse.coinBalance@pre = Set{@coin1,@coin2}`. But beside `coin1`, the balance contains `coin3` in the post state.

3.4. Exceptional scenario: world unchanged

The last object diagram shown in Fig. 8 is reached after an invalid call of `abTransferLost` in state 4. It describes a world which did not change during the operation invocation. But security properties require a decreasing balance and an increasing lost value within the *from* purse.

```
postcondition ‘FromPurseBalanceDecreases’ is false
evaluation results:
  fromPurse : AbPurse = @cainP
  fromPurse.coinBalance : Set(Coin) = Set{@coin3}
  fromPurse : AbPurse = @cainP
  fromPurse.coinBalance@pre : Set(Coin) = Set{@coin3}
  value : Set(Coin) = Set{@coin3}
  (fromPurse.coinBalance@pre - value) : Set(Coin) = Set{}
  (fromPurse.coinBalance =
   (fromPurse.coinBalance@pre - value)) : Boolean = false
```

After termination of `abTransferLost`, the object `coin3` should no longer be linked to Cain’s purse as a balance element, but as a lost coin. In this scenario the requirement is not fulfilled. The coin is still existing as a balance value which violates the postcondition `FromPurseBalanceDecreases`. The second constraint violated is `FromPurseLostIncreases` because Cain’s set of lost coins is empty in the post state although a `Lost` link should be inserted between `cainP` and `coin3`.

```
postcondition ‘FromPurseLostIncreases’ is false
evaluation results:
  fromPurse : AbPurse = @cainP
  fromPurse.coinLost : Set(Coin) = Set{}
  fromPurse : AbPurse = @cainP
  fromPurse.coinLost@pre : Set(Coin) = Set{}
  value : Set(Coin) = Set{@coin3}
  fromPurse.coinLost@pre->union(value) : Set(Coin) = Set{@coin3}
  (fromPurse.coinLost =
   fromPurse.coinLost@pre->union(value)) : Boolean = false
```

4. System evolution in the declarative approach

The declarative approach is analyzed with the same scenarios as considered in the previous sections. Figure 9 shows the corresponding sequence diagram. In contrast to the imperative approach, there is only one system state finally covering the four separate object diagrams of the imperative approach. That system state is created step by step with the four meta operations, i.e., special operations that build up the system state and roughly

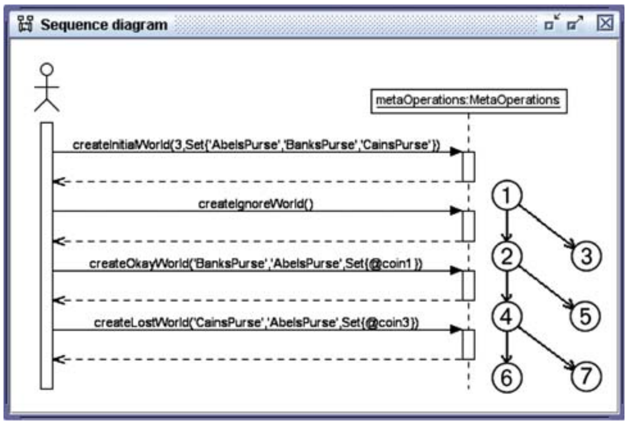

Sequence diagram
metaOperations:MetaOperations
createInitialWorld(3,Set{'AbelsPurse','BanksPurse','CainsPurse'})
createIgnoreWorld()
createOkayWorld('BanksPurse','AbelsPurse',Set{@coin1})
createLostWorld('CainsPurse','AbelsPurse',Set{@coin3})
1
2
3
4
5
6
7

Fig. 9. Sequence diagram—declarative approach

correspond to ordinary transfer operations. Before the first meta operation is invoked, the object diagram is empty. All operations create one world each possessing instances of Abel's, the Bank's and Cain's purse and connect the new world to the predecessor world. The first operation `createInitialWorld` also creates three coins. The call to `createIgnoreWorld` essentially creates the object `secondWorld`, the call to `createOkayWorld` the object `thirdWorld`, and the call to `createLostWorld` the object `fourthWorld`. After termination of this last meta operation call the object diagram, i.e., the world chain, is complete. The complete object diagram is shown in Fig. 10. Every number in the sequence diagram refers to a particular world created by the meta operation executed in the last step. The world chain represented by the numbers 1, 2, 4 and 6 describes the mainstream scenario. To inspect one of the exceptional scenarios, a world of the mainstream scenario has to be replaced by an exceptional one. In the exceptional scenarios, the second world 2 is be replaced by world 3, the third world 4 by world 5 and the fourth world 6 by world 7.

4.1. Mainstream scenario

Unlike the imperative approach where one world is created which evolves through three calls of abstract operations, the evolution is explicitly modeled here in the declarative approach. Figure 10 pictures the object diagram describing the whole mainstream scenario. There are four successive worlds with different purse objects and three coins connected with exactly one purse in every world. The purse objects represent a particular purse, e.g., `abelPInit`, `abelPSecond`, `abelPThird` and `abelPFourth` refer to Abel's purse (for these four purses we have `name='AbelsPurse'`). But the values of their `name` attribute are not displayed in this overview. The world chain models the `abTransfer` operations resulting from the meta operation calls. Every world is displayed with more details in Fig. 13.

To inspect different properties of the modeled scenario, OCL queries can be employed. The USE system evaluates them and displays their result. Figure 11 shows an expression inquiring the values transferred during the transitions induced by abstract operation calls. The variable `init` is defined to shorten the query. It refers to the initial world, `init.succ` to its successor, i.e., the object `secondWorld`, and `init.succ.succ` to the third world. The `iterate` expression takes one world after the other and includes the monetary values used within the corresponding transfer details into a sequence of sets of coins. Undefined values are included as empty sets. The result `Sequence{Set{}, Set{@coin1}, Set{@coin3}}` represents the monetary values on the transitions and shows that (1) no value is transferred during the execution of the first transfer, (2) the second transfer involves `coin1` and (3) the third transfer deals with `coin3`. Please note that the result `Sequence{Set{}, Set{@coin1}, Set{@coin3}}` accumulates a property of the world chain into a single complex value.

The world sequence `Sequence{initialWorld, initialWorld.succ, initialWorld.succ.succ}` can also be constructed with an `iterate` term in cases when there are more worlds included. The OCL query displayed in Fig. 12 will construct the world sequence in the above situation.

Figure 13 points out the four successive worlds existing in the mainstream scenario. No world exists before the meta operation `createInitialWorld` is invoked which creates the first world 1 corresponding to the initial world of the imperative approach except for the additional transfer details linked to it and the names identifying

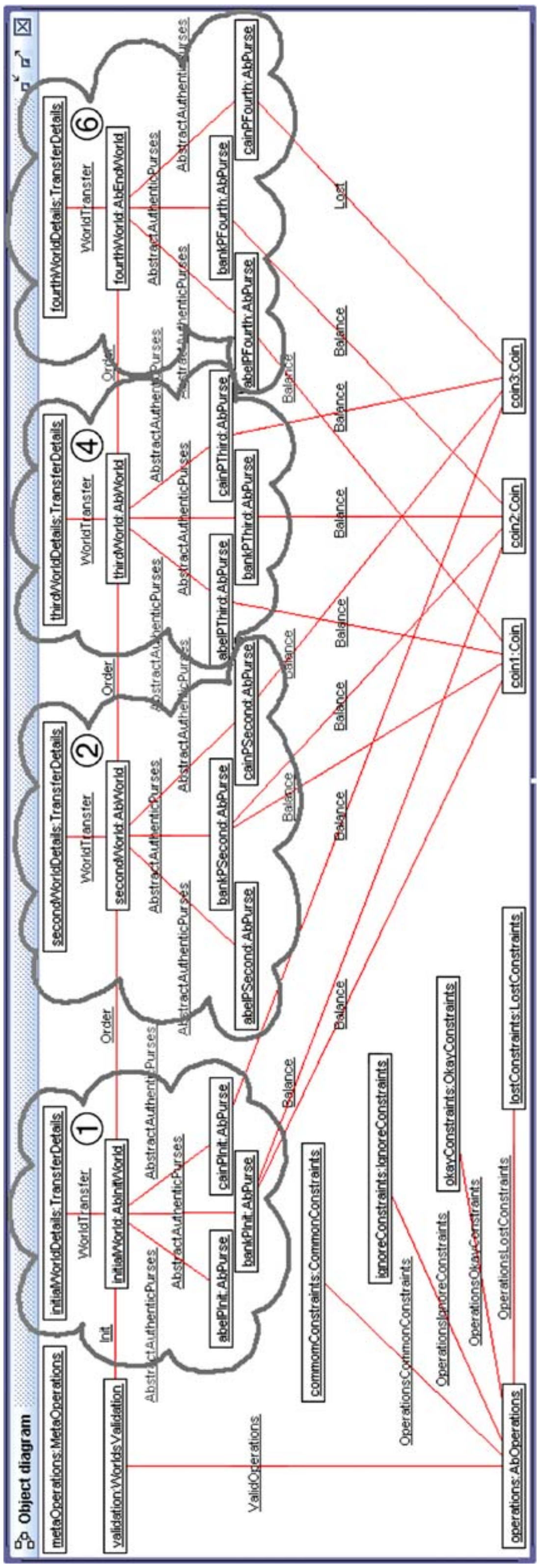
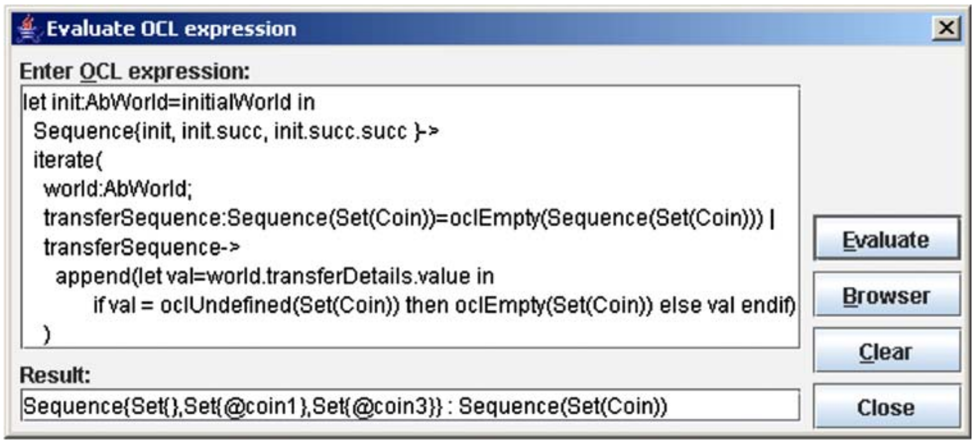


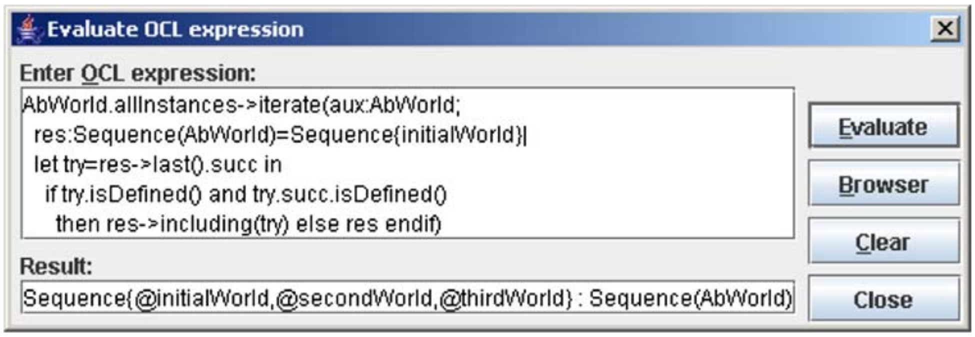
Fig. 10. Object diagram—declarative approach



```
let init:AbWorld=initialWorld in
 Sequence{init, init.succ, init.succ.succ }->
 iterate(
  world:AbWorld;
  transferSequence:Sequence(Set(Coin))=oclEmpty(Sequence(Set(Coin))) |
  transferSequence->
   append(let val=world.transferDetails.value in
        if val = oclUndefined(Set(Coin)) then oclEmpty(Set(Coin)) else val endif)
 )
```

```
Sequence{Set{},Set{@coin1},Set{@coin3}} : Sequence(Set(Coin))
```

Fig. 11. OCL query: sequence of transferred values



```
AbWorld.allInstances->iterate(aux:AbWorld;
 res:Sequence(AbWorld)=Sequence{initialWorld}|
 let try=res->last().succ in
  if try.isDefined() and try.succ.isDefined()
   then res->including(try) else res endif)
```

```
Sequence{@initialWorld,@secondWorld,@thirdWorld} : Sequence(AbWorld)
```

Fig. 12. OCL query: sequence of successive worlds

the purse objects. The attributes in object `initialWorldDetails` are undefined because the pair of worlds 1 and 2 should describe a parameterless abstract ignore stated with the operation `createIgnoreWorld` which creates the second world and connects it as successor. This world represents the same constellation of purses and coins as the initial world. The transfer details connected with the second world are defined. Its attribute values correspond to the parameters passed to the operation `abTransferOkay` within the scenario of the imperative approach. According to the transfer of balance in world 2, world 4 shows that `coin1` switched its owner from the bank to Abel. `createLostWorld` creates the last world 6 and sets the attributes of the transfer details in world 4 to values defined by the operation's actual parameters. World 4 and 6 demonstrate an abstract lost of `coin3`. The connection between Cain's purse and this coin switches from a `Balance` link to a `Lost` link. World 6 represents the end world with no successor. Therefore the attributes in the last transfer details remain undefined.

4.2. Exceptional scenario: world changes

If world 2 is replaced by world 3 in the mainstream scenario, an exceptional scenario analogously to the scenario explained in Sect. 3.2 arises (see Fig. 14). That means a new purse is created during the execution of an abstract ignore. This violation is indicated by the falsity of the invariant `ValidProgressOfTransfers`. The USE system shows its result in a *Class invariants* window shown in Fig. 15. It also shows a violation of invariant `ValidPurseNames` because the new purse has an undefined name. USE enables the analysis of all invariants with the aid of the so-called *Evaluation browser* as depicted in Fig. 16. The browser can be conveniently opened, for example, by double-clicking a failing invariant in the Class invariants window. The browser helps to inspect a constraint and all OCL subexpressions the constraint consists of. The constraint and its subexpressions are evaluated according to the available object diagram. It is possible to identify objects and links causing faulty system

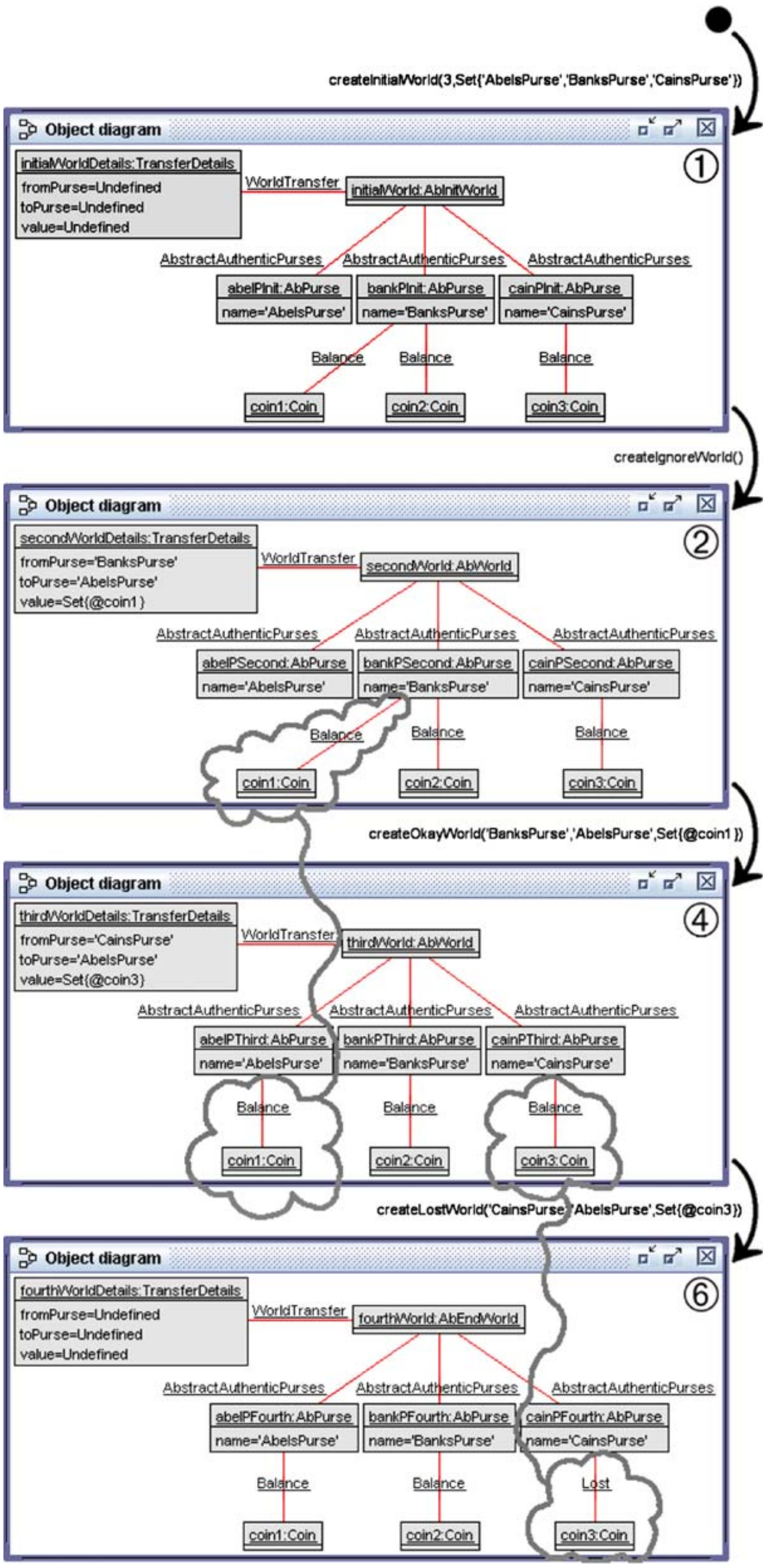


Fig. 13. Mainstream scenario of the declarative approach

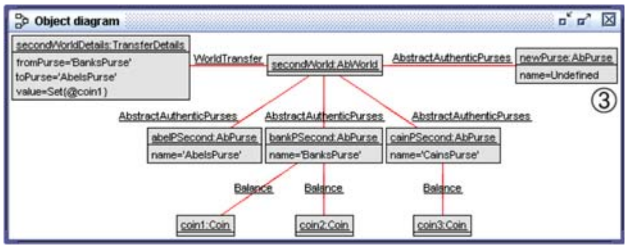
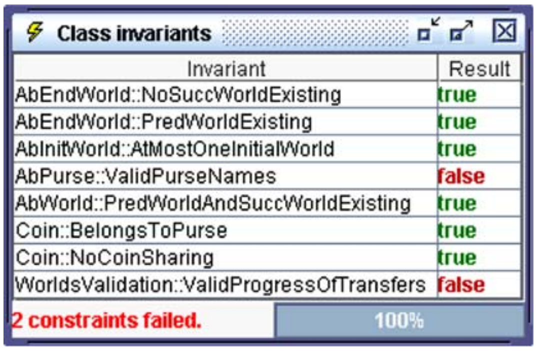


Fig. 14. Second world—new purse created


Class invariants

Invariant	Result
AbEndWorld::NoSuccWorldExisting	true
AbEndWorld::PredWorldExisting	true
AbInitWorld::AtMostOneInitialWorld	true
AbPurse::ValidPurseNames	false
AbWorld::PredWorldAndSuccWorldExisting	true
Coin::BelongsToPurse	true
Coin::NoCoinSharing	true
WorldsValidation::ValidProgressOfTransfers	false

2 constraints failed. 100%

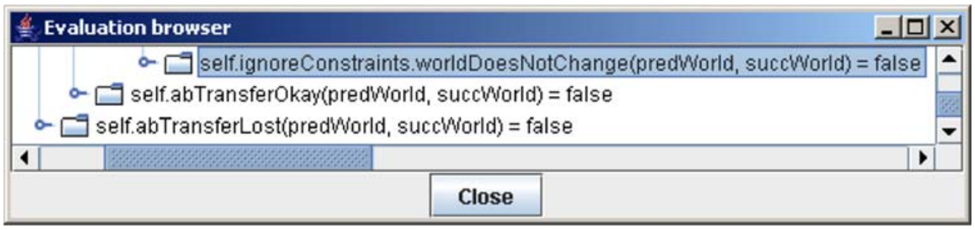
Fig. 15. Class invariants—ValidPurseNames and ValidProgressOfTransfers violated

states even in large object diagrams. The diagram modeling this scenario causes the boolean query `abIgnore` to return *false* in context of the successive worlds 1 and 3. The displayed OCL expression `self.ignore-Constraints.worldDoesNotChange(predWorld, succWorld)` represents a subexpression in form of a conjunct within the `abIgnore` definition. Below this expression, the browser shows the falsity of `abTransferOkay` and `abTransferLost` in context of the same pair of worlds. They must not be true, because an abstract ignore is simulated.

Figure 16 displays the evaluation browser indicating that the auxiliary query `worldDoesNotChange` returns *false*, which implies the falsity of `abIgnore`. It is possible to examine the operation by opening the evaluation tree to see the operation definition. For the above situation, the part of the constraint leading to *false* is displayed in Fig. 17. All purse objects existing in world 3 must have an equivalent representing the same purse in the predecessor world 1. The previous set of purses `Set{@abelPInit, @bankPInit, @cainPInit}` does not include an instance representing the new purse. That means there is no purse object which has the same name as `newPurse`.

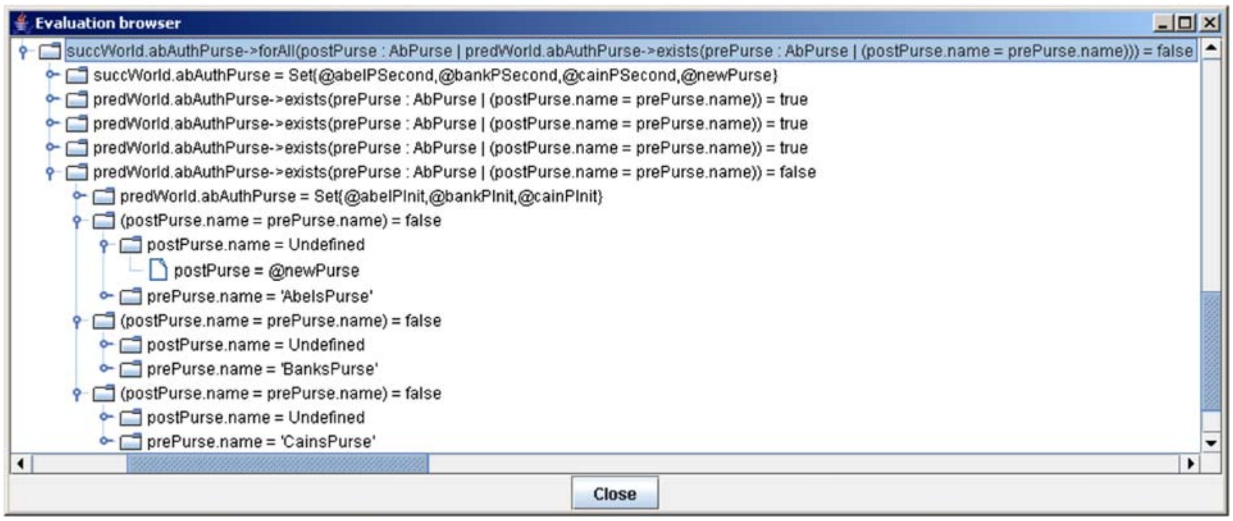
4.3. Exceptional scenario: further transfer

World 5 shown in Fig. 18 replaces world 4 in the mainstream scenario to model the exceptional scenario with a further transfer. `coin3` is transferred from Cain's purse to the bank although Cain is not involved in the transfer. The transfer details of world 2 define that only Abel and the bank should be participating (see Fig. 13). Otherwise



```
self.ignoreConstraints.worldDoesNotChange(predWorld, succWorld) = false
self.abTransferOkay(predWorld, succWorld) = false
self.abTransferLost(predWorld, succWorld) = false
```

Fig. 16. Evaluation browser—worldDoesNotChange returns false



```
Evaluation browser
succWorld.abAuthPurse->forAll(postPurse : AbPurse | predWorld.abAuthPurse->exists(prePurse : AbPurse | (postPurse.name = prePurse.name))) = false
  succWorld.abAuthPurse = Set{@abelPSecond,@bankPSecond,@cainPSecond,@newPurse}
  predWorld.abAuthPurse->exists(prePurse : AbPurse | (postPurse.name = prePurse.name)) = true
  predWorld.abAuthPurse->exists(prePurse : AbPurse | (postPurse.name = prePurse.name)) = true
  predWorld.abAuthPurse->exists(prePurse : AbPurse | (postPurse.name = prePurse.name)) = true
  predWorld.abAuthPurse->exists(prePurse : AbPurse | (postPurse.name = prePurse.name)) = false
    predWorld.abAuthPurse = Set{@abelPInit,@bankPInit,@cainPInit}
    (postPurse.name = prePurse.name) = false
      postPurse.name = Undefined
        postPurse = @newPurse
      prePurse.name = 'AbelsPurse'
    (postPurse.name = prePurse.name) = false
      postPurse.name = Undefined
      prePurse.name = 'BanksPurse'
    (postPurse.name = prePurse.name) = false
      postPurse.name = Undefined
      prePurse.name = 'CainsPurse'
Close
```

Fig. 17. Evaluation browser—details of worldDoesNotChange

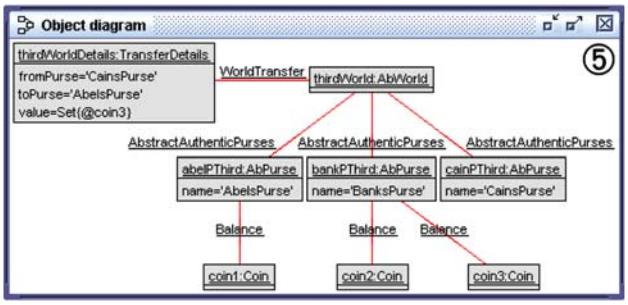
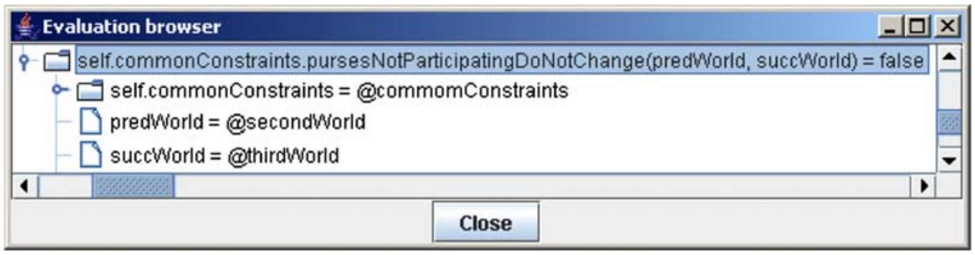


Fig. 18. Third world—coin3 transferred to bankPThird

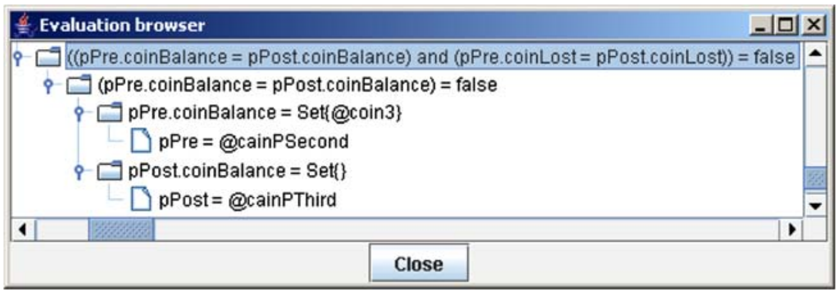
the security properties are not fulfilled. The violated property realized with an OCL query is identified with the evaluation browser (see Fig. 19).

The query `pursesNotParticipatingDoNotChange` returns *false*. It falsifies the whole boolean expression defining the query `abTransferOkay`. Figure 20 displays the part of the constraint which evaluates to false. The balance of Cain's purse in the predecessor world (`cainPSecond.coinBalance`) must equal the balance of his purse in the successor world (`cainPThird.coinBalance`), but `coin3` is no longer a balance element of Cain's purse in world 5 (`pPost`).



```
Evaluation browser
self.commonConstraints.pursesNotParticipatingDoNotChange(predWorld, succWorld) = false
  self.commonConstraints = @commomConstraints
  predWorld = @secondWorld
  succWorld = @thirdWorld
Close
```

Fig. 19. Evaluation browser—pursesNotParticipatingDoNotChange returns false



```
((pPre.coinBalance = pPost.coinBalance) and (pPre.coinLost = pPost.coinLost)) = false
  (pPre.coinBalance = pPost.coinBalance) = false
    pPre.coinBalance = Set{@coin3}
      pPre = @cainPSecond
    pPost.coinBalance = Set{}
      pPost = @cainPThird
```

Fig. 20. Evaluation browser—details of pursesNotParticipatingDoNotChange

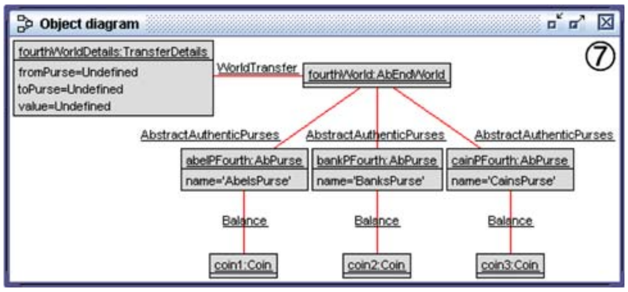


Fig. 21. Fourth world—no changes

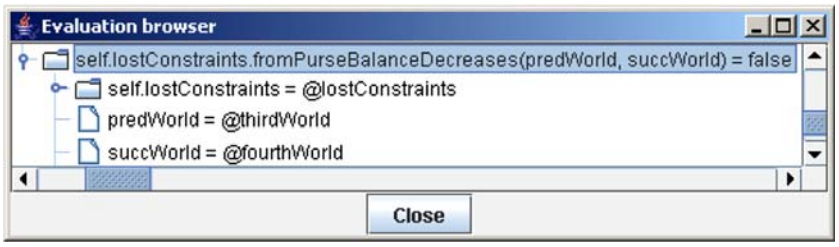
4.4. Exceptional scenario: world unchanged

The fourth world 6 in the mainstream scenario describes the result of an abstract loss of `coin3`. In the exceptional scenarios, it is replaced by world 7, which is equal to world 4 (see Fig. 21). This implies the violation of the invariant `ValidProgressOfTransfers` invoking the validation of the world chain, because the query `fromPurseBalance-Decreases` called in `abTransferLost` does not return *true* (see Fig. 22).

The details are shown in Fig. 23. Cain's balance in the fourth world (`cainPFourth.coinBalance`) should equal the balance in the third world (`cainPThird.coinBalance`) without the lost `coin3` which is the only element in the set of coins to be transferred (`thirdWorld.transferDetails.value`). Both the previous and the actual balance include `coin3`. This fact falsifies the required property.

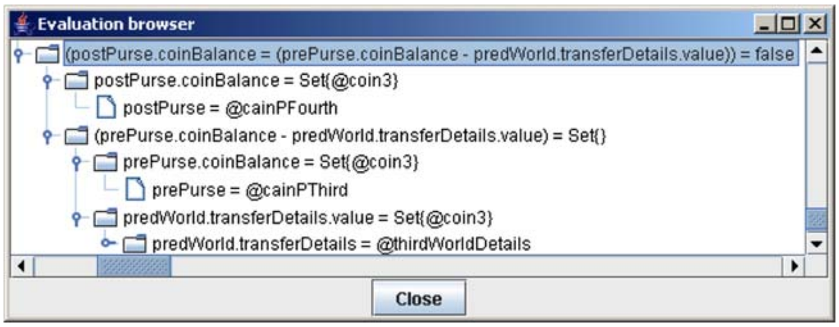
5. Comparison of both approaches

This section compares the developed imperative and declarative approach and recapitulates their central characteristics. Their advantages and disadvantages are considered in terms of UML, respectively, OCL aspects and the UML-based Specification Environment and its ability to animate and validate both models. We will treat differences in the evolution of the world, the access of the security properties as well as model complexity and understandability.



```
self.lostConstraints.fromPurseBalanceDecreases(predWorld, succWorld) = false
  self.lostConstraints = @lostConstraints
  predWorld = @thirdWorld
  succWorld = @fourthWorld
```

Fig. 22. Evaluation browser—fromPurseBalanceDecreases returns false



```
(postPurse.coinBalance = (prePurse.coinBalance - predWorld.transferDetails.value)) = false
  postPurse.coinBalance = Set{@coin3}
    postPurse = @cainPFourth
  (prePurse.coinBalance - predWorld.transferDetails.value) = Set{}
    prePurse.coinBalance = Set{@coin3}
      prePurse = @cainPThird
    predWorld.transferDetails.value = Set{@coin3}
      predWorld.transferDetails = @thirdWorldDetails
```

Fig. 23. Evaluation browser—details of fromPurseBalanceDecreases

In the imperative approach we realize the abstract operations `abIgnore`, `abTransferOkay` and `abTransfer-Lost` by declaring three side-effected OCL operations, i.e., we specify their names, formal parameters and the return types, but not their the implementations. The effects the OCL operations have on the system state in which they are invoked are defined by the use of postconditions representing boolean OCL expressions which have to be true after operation termination. Analogously OCL provides for the evaluation of preconditions constraining the possibility of calling an operation in a system state. Both pre- and postconditions do not dictate a particular implementation. The UML-based specification environment provides the special commands `openter` and `opexit` to explicitly indicate the invocation and the termination of an operation. After the invocation the user has the possibility to input commands changing the system state (`create` *object*, `destroy` *object*, `set` *attribute*, `insert` *link*, `delete` *link*). In this work, we have collected the commands in USE command files, which can be read in by the USE system.

Another way of creating command lists is the use of the automatic snapshot generator available in USE [GBR05]. The generator allows for executing so called ASSL procedures representing different implementations of an operation. They are not forced to be deterministic, i.e., they are able to try different ways of reaching the desired system state. For example different coins, respectively, set of coins may be involved in an abstract transfer. In the following we will dwell on the potentials for extending our approaches with the snapshot generator in future work. Currently there is a case study using the generator in connection with the Smartcard application [APW07].

Particularly with regard to the declarative approach the use of the generator shall give advantages. In the declarative approach we do not alter an abstract world by calling explicitly an abstract operation. The chain of successive worlds is created en bloc. We introduced the meta operations to structure the creation process. They appear to be the equivalents of the abstract operations specified in the imperative approach, especially when we compare both sequence diagrams. But the meta operations do not have pre- or postcondition and thus do not consider any security property. The first three meta operation calls in Fig. 9 do not result in valid system states. They were declared to establish an analogy to the imperative approach and to provide an insight in the actions executed during the creation processes. Due to the fact that the abstract operations and all security properties are modeled as boolean OCL queries it is not possible to invoke an abstract transfer and to inspect the validity of the resulting system state directly. In case of side-effected OCL operations the USE tool lists violated pre- and postconditions, i.e., security properties, and gives a detailed explanation why they do not hold. This is not applicable in case of OCL queries, because we have to inspect the system states via the evaluation browser, which does not present the violated parts automatically. Concerning the suitability to familiarize the clients with the Smartcard system this is a disadvantage compared to the imperative approach. By contrast, the declarative approach provides several features for more sophisticated analysis of the abstract transfers.

A complete chain of abstract worlds, including an initial and end world, is the basis for a valid system state, i.e., for a state fulfilling all OCL invariants. The invariant `WorldsValidation` checks all security properties with respect to all pairs of successive worlds at once. According to this the evaluation browser shows all OCL expressions and subexpressions, which were evaluated in the context of the inspected chain of worlds and the selected invariant. The evaluation trees are large, because they show both fulfilled and violated security properties. Thereby the user can also retrace the reason for a positive evaluation result. However navigating in large evaluation trees demands a certain understanding of the underlying specification and its structure, which is more complex than the specification of the imperative model, because of the explicit modeling of predecessor and successor worlds.

The moderate amount of added complexity is tolerable in view of the resulting merits. The declarative model is appropriate for an extensive use of the snapshot generator, which checks the invariants after executing an ASSL procedure. A violated invariant results in a further execution of the procedure following another way of changing the system state. Our future work will include the specification of a parameterized ASSL procedure, which accepts arguments like the length of the chain to be created, the maximum number of allowed abstract ignores, the number of purses, respectively, coins, and other conditions, which have influence on the resulting chain of abstract worlds. The generator will create random chains with random transfers between the successive worlds. In the imperative approach the user has to manually choose the abstract operation to be invoked, which is a considerable limitation.

Other than that mentioned features the snapshot generator can be used to check whether under the given conditions, i.e., parameters of the procedure, there is actually a sequence of abstract transfers resulting in a valid state. Besides analyzing each security property individually the generators enables the examination of implicit constraints resulting from the conjunction of all security properties. For example if we require an ASSL procedure to create a system state including at least three calls of the operation `abTransferLost` and additionally permit the existence of at most two coins in an abstract world, the generator will provide the answer that no valid state was found and how many different system states were reached during execution.

In conclusion both approaches serve different purposes. Nevertheless they belong together. The imperative model has a simple class diagram which includes the basic elements of the Mondex system. In USE the abstract operations can easily be animated and the possibly violated pre- and postconditions have meaningful names. After being acquainted with the principles of the electronic purses clients can use the declarative approach for further insights. The class diagram associated with the declarative model makes the workflow of the Mondex system more explicit and gives an overview of the available security properties and their roles. The meta modeling approach induces a large and in some cases a slightly confusing object diagram. However whereas the object diagrams in the imperative model are meant to be displayed and visually inspected, the declarative approach aims for more textual analysis.

6. Conclusion

In this paper we have described the abstract model of the Mondex case study with UML and OCL and have animated central aspects with the USE system (UML-based Specification Environment). Our main motivation and aim was the explanation of the complex formal requirements in an easy-to-understand way.

We have shown the overall structure of the imperative model and the declarative model with UML class diagrams. Restricting constraints were specified with OCL. We have used UML object diagrams within USE in order to explain typical system states. UML sequence diagrams represented in USE the different scenarios and the temporal system development. We have shown positive and negative test cases. The USE evaluation browser helped in analyzing errors, and OCL queries made it possible to inspect the object diagrams and to detect system properties.

Apart from the concrete Mondex model we could develop analogously the concrete model with UML and OCL. However, no formal refinement notion is currently available in OCL. One solution could be to include the abstract description and the concrete description in one USE model. Then one could explicitly connect respective entities with associations or with constraints, e.g., an association could be established between the abstract purse and the concrete purse. One could formulate the refinement requirements as constraints in the context of the connecting associations (association classes) or with other OCL constraints. Analogously to other approaches like VDM, Z, or Alloy, one would have then to prove that the concrete model is a refinement of the abstract model. This means to prove that the abstract and the concrete model together imply the refinement requirements. Tool support for this is underway because first steps towards proof systems for OCL are currently under development [KFdB^{+}05, BW02].

References

[APW07] Aydal E, Paige R, Woodcock J (2007) Evaluation of OCL for large-scale modelling: a different view of the Mondex smart card application. In: Giese H (ed) Satellite events at the MoDELS'2007 conference. LNCS, 2007. OCL workshop'2007. Springer, Berlin

[But06] Butler M (2006) The Mondex case study specified in B. In: Bicarregui J, Woodcock J (eds) 1st Mondex workshop. http://qpq.csl.sri.com/vsr/private/repository/MondexCaseStudy

[BW02] Brucker AD, Wolff B (2002) HOL-OCL: experiences, consequences and design choices. In: Jézéquel J-M, Hussmann H, Cook S (eds) UML 2002: model engineering, concepts and tools. LNCS, vol 2460. Springer, Heidelberg, pp 196–211

[CPC$^+$04] Chiorean D, Pasca M, Cârcu A, Botiza C, Moldovan S (2004) Ensuring UML models consistency using the OCL environment. ENTCS 102:99–110

[Cro06] Crocker D (2006) The Mondex case study specified in perfect developer. In: Bicarregui J, Woodcock J (eds) 1st Mondex workshop. http://qpq.csl.sri.com/vsr/private/repository/MondexCaseStudy

[GBR05] Gogolla M, Bohling J, Richters M (2005) Validating UML and OCL models in USE by automatic snapshot generation. J Softw Syst Model 4(4):386–398

[Gro05] Object Management Group (2005) OMG unified modeling language, version 2.0. OMG, http://www.omg.com/uml

[JAB$^+$06] Jouault F, Allilaire F, Bézivin J, Kurtev I, Valduriez P (2006) ATL: a QVT-like transformation language. In: Tarr PL, Cook WR (eds) OOPSLA companion. ACM, New York, pp 719–720

[Jon06] Jones C (2006) The Mondex case study specified in VDM. In: Bicarregui J, Woodcock J (eds) 1st Mondex workshop. http://qpq.csl.sri.com/vsr/private/repository/MondexCaseStudy

[KFdB$^+$05] Kyas M, Fecher H, de Boer FS, Jacob J, Hooman J, van der Zwaag M, Arons T, Kugler H (2005) Formalizing UML models and OCL constraints in PVS. ENTCS 115:39–47

[KG08] Kuhlmann M, Gogolla M (2008) Complete USE sources for the Mondex case study. University of Bremen, Bremen. ftp://ftp.informatik.uni-bremen.de/local/db/papers/fac2008

[RG01] Richters M, Gogolla M (2001) OCL—syntax, semantics and tools. In: Clark T, Warmer J (eds) Advances in object modelling with the OCL. LNCS, vol 2263. Springer, Berlin, pp 43–69

[RJ06] Ramananandro T, Jackson D (2006) The Mondex case study specified in alloy. In: Bicarregui J, Woodcock J (eds) 1st Mondex workshop. http://qpq.csl.sri.com/vsr/private/repository/MondexCaseStudy

[RJB05] Rumbaugh J, Jacobson I, Booch G (2005) The unified modeling language reference manual. Object technology series, 2nd edn. Addison-Wesley, Reading

[SCW00] Stepney S, Cooper D, Woodcock J (2000) An electronic purse: specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, New York

[SGHR06] Schellhorn G, Grandy H, Haneberg D, Reif W (2006) The Mondex challenge: machine checked proofs for an electronic purse. Technical report 2006-02, Institute of Computer Science, University of Augsburg

[WK03] Warmer J, Kleppe A (2003) The object constraint language: getting your models ready for MDA. Addison-Wesley, Reading

[Woo06] Woodcock J (2006) The Mondex case study specified in Z. In: Bicarregui J, Woodcock J (eds) 1st Mondex workshop. http://qpq.csl.sri.com/vsr/private/repository/MondexCaseStudy

Received 19 January 2007

Accepted in revised form 9 November 2007 by J. C. P. Woodcock

Published online 12 December 2007