

Generating OCL Specifications and Class diagrams from Use Cases: A Newtonian Approach

Boris Roussev

*Accounting and Information Systems Department
Sigmund Weis School of Business
Susquehanna University
broussev@acm.org*

Abstract

The universal adoption of UML for object-oriented modeling notwithstanding, a major impediment for UML 1.4 is the lack of seamless integration between the different models and the inadequate support for diagram interchange. In this paper we propose a process generating formal object-oriented specifications in OCL and class diagrams from the use case model of a system through a clearly defined sequence of model transformations. The algebraic invariant of values exchanged in a use case guides the derivation of state machine descriptions for the actors and counter-actors, collectively called agents of the use case. The use case specification is obtained as the composition of the agents' state machines. We present an algorithm for converting a narrative use case specification to a set of OCL expressions and/or class diagrams. The proposed approach fills the gap between the outside behavioral system description as offered by use cases and the "first cut" at software architecture, the analysis level class model.

1. Introduction

The contention of Ivari [15] and Wieringa [29] is that object-oriented methods are underdeveloped in the specification of external functions of systems and weak in guidelines to partition a system into subsystems. This assertion is supported by the empirical data provided in the CHAOS research report by the Standish group [26]. The study maintains that almost three out of four software projects fail in one or more of the following areas: exceeded deadlines, exceeded budgets, faulty solutions or unmaintainable systems. The foremost question is: What causes software projects to fail?

One view maintains that computing science has not "matured" from a theoretical topic for the scientists to a practical issue for the engineers [11, 23]. We do not have our $F=ma$ (or we don't need one) [2]. We lack first

principles and therefore cannot distinguish between good and bad architectures [7].

The work of Jacobson [16] went a long way in bringing the craft of software development closer to the family of engineering disciplines. Jacobson's approach for software development was embraced by the OMG and evolved into UML [5]. UML is an object modeling language for specifying, constructing, visualizing, and documenting the artifacts of software-intensive systems [22]. UML is used to model systems from a variety of views at different level of abstraction. A UML model of a software system consists of several partial models, each of which addresses a certain set of structural, behavioral, communication or interaction issues.

The success of Jacobson's software development process was predicated on three factors: (1) driven by user requirements; (2) designed to contain "changeability" through "locality of change"; and (3) the realization that the huge semantic gap between the problem and the machine domain cannot be bridged by a single model. As a result, the software development process is defined as model transformation from more abstract to more detailed and precise models.

What makes a good model or metaphor? In his research on psychoanalysis Sigmund Freud was inspired by poetry and myths. The classical metaphor example "My love is a red rose" has a rich semantic potential. The poet compares two seemingly disparate things (we call this modeling) to convey that love is beautiful, passionate, tender, non-possessive, requiring nurture and care, cannot be taken for granted, but love is also bound to bring pain, for roses do have thorns. Note the level of abstraction. The inessential detail is suppressed. Whether it is Dulcinea Del Toboso or Don Quixote is inconsequential. Note the range of meaning only by two words! This is precisely what we, software developers, need — modeling languages with rich semantics and expressive power — and in all fairness, we do not yet have them to hand.

A serious disadvantage of UML is its semiformal

syntax and semantics. Modeling from different viewpoints, successive refinements and the absence of formal semantics give rise of vertical, horizontal, syntactic and semantic consistency problems [13]. The different models must be consistent and uncontradictory for an implementation to be feasible.

Our objective is to facilitate the development of UML behavior models, class diagrams and formal OCL specifications from system requirements expressed as use cases. The main contributions of this paper enhance Jacobson's first and third factors of success, namely use case specification and model transformation. We propose the use of algebraic invariants as a basis for discovering classes, class relationships and OCL constraints from use cases. The rigorous approach is based on the state machine model and fills the gap between the outside behavioral system description as offered by use cases and the "first cut" at system architecture, the analysis level class model. We show how use cases, state machines and class diagrams seamlessly relate to each other.

The advantage of adding rigor to the use case model and to the process of identifying classes and class relationships from the formalized use case model is, in Dijkstra's spirit, increased use of calculation at the expense of "liberal creativity," while in poetic terms it creates a richer semantic meaning for a model/metaphor.

The remainder of the paper is structured as follows. In Section 2, we present the UML elements used in this work and review the notion of value added invariant. In Section 3, we introduce informally a process generating the software architecture of a system from its use case model based on the value added invariant. Next, in Section 4 we give the algorithm describing formally the proposed approach. In the final section, we outline plans for future work and conclude.

2. UML models used and UML critique

2.1. Use cases

A *use case* is a sequence of transactions performed by an actor in a dialogue with the system that brings value to the actor [16]. Cockburn observed in [9] the confusion resulting from the different perception of use cases and gives 24 possible (many of them valid) interpretations of the use case notion. For a most efficient comprehension of the difficulties associated with use case interpretation, we adopt a framework for system development similar to [29]. A system is an aggregation of interacting subsystems serving a useful purpose. A system delivers a service to its environment by interacting with it. Interactions can be partitioned into pieces that bring some value to the environment, called functions. System interactions are by nature communications with one or more external entities in the environment of the system. The order of the functions in time is called behavior of the

system. We can describe system properties (functions, communications and behavior) at different levels of abstraction. This sets up the notion of refinement hierarchy.

To develop a system, we must specify the desired external interactions, then specify a decomposition, and finally allocate external interactions to component interactions. We are primarily interested in conceptual decomposition, which is a partitioning into conceptual components in terms of their meaning to the external environment (actors in UML). The conceptual decomposition bridges the gap between the external functional specification (requirements) and the internal physical decomposition (design). In the UML community, there is an overall agreement that system functions are rendered as use cases, system decomposition is represented by class diagrams, component behavior by statecharts, and component communication by sequence or collaboration diagrams.

Use cases, however, do not capture only functional requirements. They are also a communication specification technique since they show external communications (with actors). In addition, use cases specify system behavior as sequences of actions. And finally, through realization relationships (an abstraction dependency being stereotyped <<realize>>) with collaborations (use case realizations), use cases indirectly determine the structural part of the analysis model as well, i.e. the static decomposition of the system. Therefore, use cases directly or indirectly express functional, behavioral, communication and structural system properties.

The Use Cases package is a subpackage of the Behavioral Elements package in the UML metamodel layer [22]. It defines elements, e.g. *UseCase* and *Actor*, used for the specification of behavior of a system or subsystem without revealing its internal structure. In the metamodel, *UseCase* is a subclass of *Classifier*. A use case realization in terms of classes, interfaces, and other classifiers can be defined with a *Collaboration*. We have found only the following pragmatic guidelines for defining use cases in the semantic description of the Use Cases package [22, Sect. 2.11.4]. Each use case specifies a service that the system provides to its users. A user initiates the service. The service must be a complete sequence, meaning that after its performance the system will be in a state in which the sequence can be initiated all over again. The interaction between a user and a use case instance continues until the instance has responded to all input and has ceased to expect any more input. Each use case should yield some value to (at least) one user (actor).

The aforementioned guidelines are rather vague. Developers can quickly fall in the trap of functional decomposition, a problem called "circled wagon" formation [4]. The major issue with a set of disconnected

functions is that they result in a hard-to-use system that does not meet the user expectations.

We can divide the use case description techniques into imperative and declarative. The advantage of an imperative specification is that it leads to an executable specification that can be validated at an early stage in the development. Its disadvantage is that it specifies a process by giving an implementation for it. The advantage of a declarative specification is that it is completely implementation-independent. However, its disadvantage is that it is an underspecification. Imperative descriptions include plain text, activity graphs and state machines, while declarative techniques are not given in the UML specification. In this work, we propose OCL pre- and postconditions over business objects to describe the effect of a use case. We also introduce a new imperative technique describing the behavior of a use case as a set of CSPs [20].

2.2. State machines and critique of diagrams modeling system's dynamics

The UML elements described in this section are used to model dynamic aspects of systems.

The state machine model formalizes the changes to the system state. A precise definition of state machine is given in Section 4. State machines can be composed, minimized, and visualized.

The UML specification describes activity diagrams as a mechanism to capture business workflows and use case flow execution. Activity diagrams describe the system decomposition into activities. Their primary focus is on the sequence and the conditions for the activities, rather than on determining which classifiers perform those activities. In that respect, they bring a little insight into structural system decomposition. Owing to their ill-defined connection with the other UML diagrams, activity diagrams pose horizontal consistency problems. Being limited in their expressiveness has been critiqued [27]. Since activity diagrams obliterate system states, their use in analysis has been, in our view, unwarrantedly promoted. Nonetheless, activity diagrams are intuitive for domain experts, and can model concurrency.

In analysis, sequence and collaboration diagrams are useful for discovering and/or verification of the software architecture. They, too, have been heavily criticized [27]. These diagrams can be neither decomposed nor composed and they do not relate to each other.

During analysis, the use case model is transformed into analysis level model with stable software architecture. A practical way of working [17] is to read through the description of each use case in a selected subset of high priority use cases and determine what classes (classifiers) and relationships are needed to realize the use case.

There are five basic approaches to class discovery from functional system specification described in the

literature: CRC [30], Noun-phrase, Common class pattern, Use case driven [16], and Mixed [18]. For a good discussion on the pros and cons of each approach see [3, 18]. The basic disadvantage of all five methods is the lack of pragmatic guidelines that can steer the process of class discovery and serve as litmus test for the quality and completeness of the resulting software architecture. Only the use case driven approach through sequence diagrams can verify the requirements, but the lack of rigor makes the verification highly subjective and it can lead to imbalance between iterative and incremental - thrashing.

Several methods combine formal specification languages (SDL, E-Lotos, Object-Z, Temporal logic) with an object-oriented method [8, 12, 21]. They aim at formalizing an inconsistent and incomplete set of requirements using as an input the sequence diagrams for a use case. The use of sequence diagrams as input already implies that the software architecture has been defined (at worst incompletely). Our approach aims at discovering the software architecture, in particular the CDs. In this respect it complements [8] and [21].

2.3. OCL

A *constraint* is a semantic restriction on one or more model elements. Types of constraints include but are not limited to constraints on associations between classes, pre- and post-conditions of class operations, multiplicity of class instances, type constraints on class attribute values, invariants on classes and guards in STG. OCL is a formal and pure expression language used to specify constraints for UML models. It combines first-order predicate logic with a diagram navigation language [22, 28]. Each OCL expression is written and evaluated in the context of (the instances) of some model element. OCL provides operations on sets, bags and sequences to support the manipulation of collections of model elements.

The semiformal semantics of UML allows for ambiguous, incomplete and inconsistent models [14]. OCL expressions augment graphical models, like class diagrams, to produce unambiguous and precise system descriptions. OCL is advantageous for a variety of reasons: (1) OCL enables a better documentation. Visual models define some constraints, like association multiplicity, but in OCL we can specify richer ones. Diagrams are incapable of conveying important elements such as uniqueness constraints, formulae, limits and business rules; (2) OCL constraints improve precision and communication; (3) Development with OCL becomes design by contract. The basic downsides of OCL are its poor readability and inefficiency in specifying requirements-level and analysis-level concepts [1]. We show in Section 3 that OCL can be used very productively in expressing analysis-level concepts.

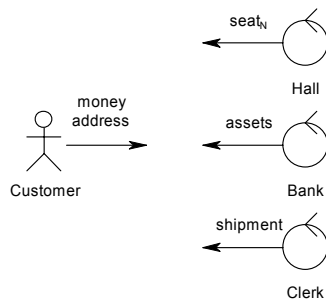


Figure 1. The agents for Book Ticket use case.

2.4. Value added invariant

In this section we review the notion of value added invariant introduced in [25].

From the perspective of an actor, a use case performs something that adds value to the actor, such as calculating a result, generating a new object or changing the state of another object [3]. Isaac Newton observed that contact forces are always exerted by one body (initiator) on another body (respondent). If we imagine that the actor initiating the use case instance is the body exerting the contact force, then there must be at least one respondent on the receiving end.

Definition 1 The respondent trading values with the actor initiating the use case instance is called *counter-actor class*, or *counter-actor* for short. Actors and counter-actors are called collectively *agents*.

For each use case there exists a conservation law, i.e. an invariant.

Axiom 1 The joint distributed count of values exchanged between the actors and the counter-actors in a use case is constant. This property is called the *value added invariant* of a use case.

The invariant is instrumental in identifying classes and OCL constraints. Jacobson gives two criteria helpful in determining the scope of a use case in [16, 17], the key phrases being “resulting value” and “participating actor.” He went a step further by saying “In some cases, actors are willing to pay for the value returned” but stopped short of generalizing and defining the exchange of values.

Example 1 Book Ticket use case: Customers with HTML 4.0 compatible browsers can book tickets online for a selected performance (from Susquehanna University Artist Series).

Customer is the actor initiating the execution of the use case. The value gained by this actor is a ticket. The value lost is some form of money. Using the invariant we discover that the counter-actors are Hall, Bank and Clerk. Hall loses one empty seat, Bank gets the

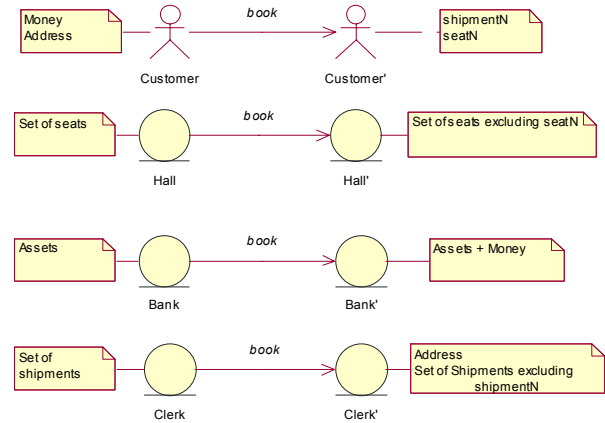


Figure 2. Value added invariant for Book Ticket.

money and Clerk gets the address value and ships. Clerk may be defined as an actor (if not already defined). The agents for the use case in the form of forces are shown in Figure 1.

At the highest level of abstraction, we define the evolution of an agent as a contract. The values lost/gained by an agent are described in the notes attached to the agent on the left/right hand side, see Figure 2. These values specify the state of an agent before/after the execution of a use case.

3. Deriving class diagrams and OCL specification from a use case

In this section we describe a process for deriving CDs and OCL constraints from a use case by structuring the use case with a set of state machines.

3.1. Describing use case as a set of state machines

Having identified all agents of a use case, we can describe, in greater detail by a state machine, the evolution each of them undergoes. The state machine model can represent a system at an arbitrary level of detail. In analysis it is mandatory that we stay at a high level of abstraction, that is out of design. This requirement is met by associating each state with a business object that is gained or lost by the action of the transition leading to the state. Figure 3 shows the state machine for the happy scenario for Customer.

To each agent gaining a value (business object) corresponds an agent losing that value. For convenience sake, we have numbered the states in each diagram so that they can be referred to unambiguously from other diagrams, if need be. The reference is in the form of agent_name[number], where agent_name is the name of an actor or a counter-actor and number is a

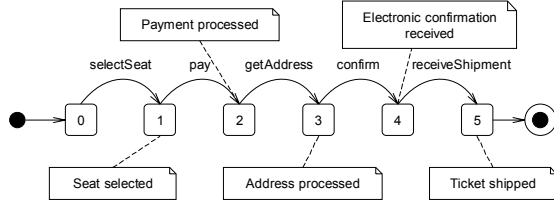


Figure 3. State machine for Customer.

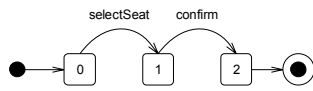


Figure 4. State machine for Hall.

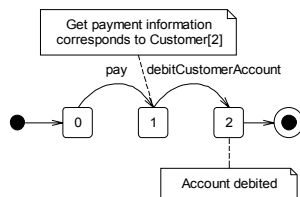


Figure 5. State machine for Bank.

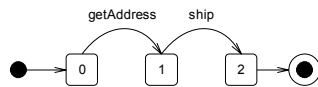


Figure 6. State machine for Clerk.

valid state index in the diagram of agent_name. The reference can be put in a note attached to the referring state, e.g. Customer[2] in Figure 4 below. The diagrams for Hall and Bank are shown in Figure 4. State numbering is optional.

3.2. Reduction of the set of state machines to a class diagram

The parallelogram law for the composition of forces was stated by Newton, see Figure 7. F_1 and F_2 are called concurrent forces. R is called the resultant and it is equivalent to the combined action of the two forces. Petri [24], Milner [20] were the first to apply the parallelogram of forces in computing science. We follow the ideas of this rich tradition to compose the static structure of the software architecture from the concurrent state machine descriptions.

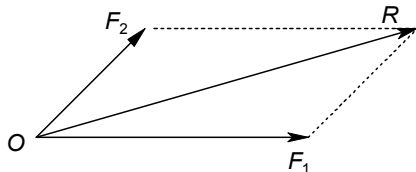


Figure 7. Parallelogram of forces.

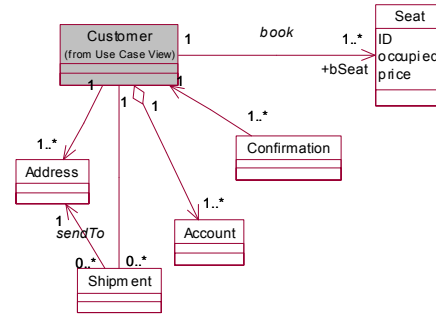


Figure 8. CD generated from the state machine model of Customer.

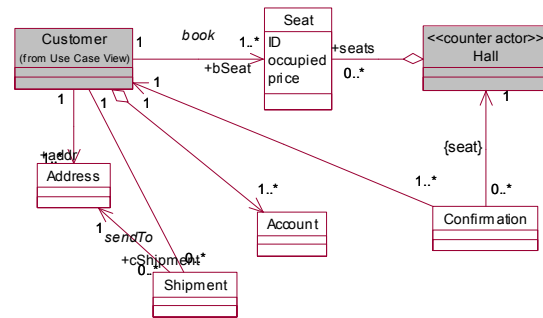


Figure 9. CD from Figure 8 extended with knowledge from the state machine model of Hall.

Before we show the composition operation we need to make a digression to introduce a result, which we will later use. Maciaszek observed that there is an interesting dichotomy with regard to actors [18]. Actors can be both external and internal to a system. They are external because they interact with the system from the outside. They are internal because the system may maintain information about actors so that it can knowingly interact with the external actors. Hence, the specification needs to hold two models related to actors — a model of the actor and a model of what the system records about the actor.

This observation leads to a heuristic for discovering of entity classes. Example 1 is a case in point. The system must create an object of class Customer (if the object already exists it is linked) for each ticket booking.

It is important to note that Jacobson did define actors as stereotyped classes [16]. However, to the best of our knowledge, the definition is for the purposes of generalization and instantiation as well as for uniform notation, e.g. the Manager actor who inherits from the Clerk actor can initiate instances of all use cases that a Clerk can and in addition some others. Actors are meant to model, like context diagrams, communication between the system and its environment.

The agent state machines are considered one at a time. We select a state machine not processed yet. We create a class for the state machine's agent. This class is being

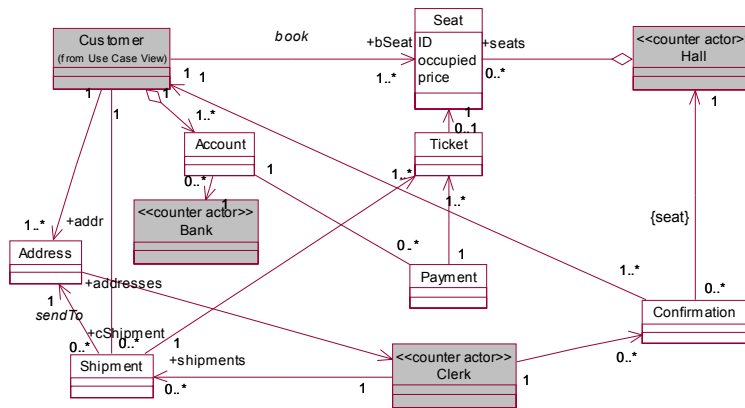


Figure 10. Final CD for Book Ticket use case.

stereotyped accordingly and shown in shaded color in Figure 8. Then we create one class for each value gained or lost by the agent. If the value is lost, we draw a unidirectional association link from the agent class to the value class. If the value is gained, the direction of the association link is opposite. Finally, we determine and add the associations creating the collaboration paths between classes necessary for the use case execution, e.g. association *sendTo* in Figure 8.

Having a rigorous approach for discovering associations among classes is important because they support the execution of use cases and, therefore, tie together the state and behavior specification.

Figure 9 shows how as a result of analyzing the state machine model of Hall we extend the initial CD with a new class and two association links. Domain knowledge suggests the use of bi-directional relationship between Hall and Seat. The CD after processing agent Clerk is presented in Figure 10.

We stereotyped the agent classes we identified as entity classes. In this way we record the long-lived changes in the redistribution of values in the system among the active participants during the performance of a use case. Since each agent interacts with other agents we duplicate the agent classes and the clones form a pool of initial candidates for control classes in the BCE approach [10]. Such a decision is justified by the behavioral aspects the agents captured by the state machine models and the existence of natural and minimal collaboration paths between the agents and the business objects.

3.3. Converting the state machines to OCL specifications

In this section we present a technique for deriving declarative use case specifications from the agent state machine models. We propose the use of OCL pre- and postconditions over business objects to describe the effect

of a use case. This is in compliance with the semantics of UML. According to the UML specification, a use case can be described using operations and methods [22].

Let us consider again Example 1 and more specifically the state machine for agent Customer in Figure 3. We define declaratively the behavior of use case Book Ticket from Customer's point of view with OCL expressions as a contract. The contract is an exact specification of the service provided by the use case w.r.t. Customer. The service is described by two sets of constraints whose context is (an instance of) Customer: (1) preconditions: the conditions under which the service will be provided; and (2) postconditions: a specification of the result, given that the preconditions are fulfilled. The service pre- and postconditions are described in the notes attached to the agents on the left and right hand side of Customer and Customer', respectively, see Figure 11(a). We split complicated constraints into several separate constraints to improve their readability and writeability (and diagnostic power in case runtime checking is employed).

The precondition for a customer to book a seat is to have a valid credit card, account for short, and a mail address coinciding with the account's mail address. The latter condition cannot be expressed with a graphical model. The postcondition of *book* from Customer's point of view includes an occupied seat, modified account balance and shipment. In addition, we define the following two constraints. *self.seat->isEmpty* is a precondition indicating that an instance of Customer has not booked a seat. The constraint treats the instance of Seat as a collection and uses the predefined property *isEmpty* of the OCL type Set. The second constraint is a postcondition using the predefined feature of all OCL types *allInstances*. It is instrumental not only in specifying a uniqueness constraint but also (implicitly through the logic *and* of all postconditions) in expressing

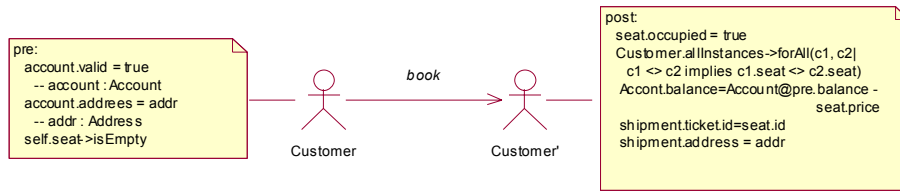


Figure 11(a). OCL specification for agent Customer.

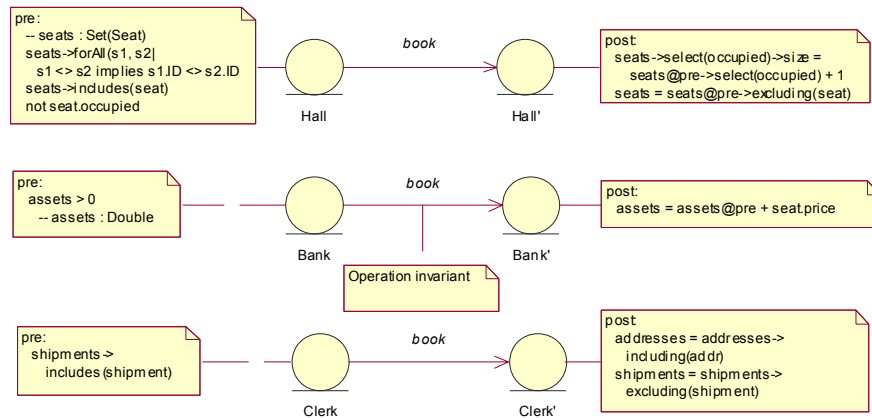


Figure 11(b). OCL specification of agents Hall, Bank and Clerk.

Figure 11. Value added invariant expressed in OCL for the Book Ticket use case.

a semantic relationship between the value of `seat.occupied` and the association `Customer-Seat`. This expression accesses the meta-level UML model to express the fact that a seat is booked by exactly one customer if and only if the seat is occupied.

Similarly to Customer, we can derive the OCL specifications for the other three agents involved in Book Ticket, see Figure 11(b). The OCL expressions are defined and evaluated in the context of the corresponding agent. `Set(Seat)` is used in the comment to explain the type of the collection seats. We use the extended variant (with two iterators) of `forAll` to define the uniqueness constraint on seats.

Declarative specifications may suffer from the so-called frame problem [6]. We can put frame assumptions in the form of invariants attached to the edges, e.g. the link `book` between Bank and Bank'. The invariant below specifies that the bank assets at any one moment are equal to the sum of the sold tickets' prices.

```
assets=select(seat | seat.occupied=true) ->
  iterate(s:Seat; result:Double=0 | result
  + s.price)
```

We use the `iterate` operation to add up the prices of all seats sold. When the `iterate` expression is evaluated, element `s` iterates over the collection of sold seats. The expression `result + s.price` is evaluated for each element `s`. After each evaluation of the expression, its value is assigned to `result`. Alternatively, the invariant can be written using the operation `collect` as follows,

```
assets=select(seat | seat.occupied=true) ->
  collect(price) ->sum
```

A final note about Figure 11 is in order. The evolution of each agent, e.g. Customer, is described by two consecutive frames (snapshots) of the agent's high-level occurrence graph. Recall from Section 2.1 that the service a use case provides must be a complete sequence of actions after the performance of which the system will be in a state in which the sequence can be executed all over again. This means that the initial state, the state defined by the use case preconditions, is a home state. Such systems are called reversible (the word re-initializable is also used).

4. Algorithm

In this section, we give an algorithm, formalizing the ideas and the method presented in section 3. The conversion formalized and described by the algorithm is done manually at this point. The complete automation of the conversion process is a topic for future research.

Let U be the use case model of a system. The set of values exchanged in the use cases $u \in U$ is called the *gain-loss set* and is denoted by V . Let A_u be the set of agents in a use case u and $V_u \subseteq V$ the gain-loss subset exchanged by the agents in u .

Definition 2 The state machine for an agent a , $a \in A_u$, is the ordered quadruplet $sm_a = (S_a, T_a \subseteq S_a \times S_a, V_a, gl_a)$, where S_a is the set of states, T_a is the set of transitions, $V_a \subseteq V_u$ is a set of values and gl_a is the *gain-loss function* defined as,

$$gl_a : W \times V_a \rightarrow T_a$$

where W is the set of real numbers in the range $[-1,1]$. gl assigns to each transition t in T_a a value v from V_a weighed by w , $w \in W$.

Notation	Meaning
U	Use case model
u	Use case
A	Actor or counter-actor
A_u	Agents in use case u
V	Loss-gain set of the model U
V_a	Loss gain subset for agent a
V_u	Loss gain subset for use case u
Act_u	Set of Actors for use case u
W	Real numbers in the range $[-1,1]$
sm_a	State machine for agent a
SM_u	Set of state machines for A_u
E_u	Set of OCL constraints for a u.c.
E_{pre}	Set of OCL preconditions for a u.c.
E_{post}	Set of OCL postconditions for a u.c.
E_{inv}	Set of OCL invariants for a u.c.
e_v	OCL expression over value v

Table 1. Notation used in the algorithm.

The algorithm generating CDs and OCL specifications from use cases is shown in Figure 12. We use the notation shown in Table 1. The use case model has n use cases. The algorithm has n iterations, one for each use case in the model. Procedure `reduce_uc_to_sm` takes a use case u , the actors Act_u of u , the actor a initiating u and the lost-gain set V (discovered thus far) as an input and returns the set of state machines SM_u for the agents of u . The lost-gain set is updated with the new values

```

procedure generate_cld_oclspec_from_ucm(
    U: use case model)
1)  $V = \emptyset$ 
2) forall usecases  $u \in U$  do begin
3)    $SM_u = \text{reduce\_uc\_to\_sm}(u, Act_u, a, V)$ 
4)    $V_u$  -- the loss-gain set of  $u$ 
5)    $V = V \cup V_u$ 
6)    $\text{reduce\_sm\_to\_cld\_oclspec}(SM_u)$ 
7) end

```

Figure 12. Procedure `generate_cld_oclspec_from_ucm`.

discovered while converting the use case to a set of state machines. Procedure `reduce_sm_to_cld_oclspec` reduces the set of state machines for the use case under consideration to one or more CDs and derives the use case's OCL specification.

Procedure `reduce_uc_to_sm` is shown in Figure 13. In the initialization phase, V_u is set to the empty set and A_u is set to the initiating actor a passed as an input to the procedure. In each iteration, procedure `generate_sm` takes an agent as an input and generates its state machine as the output. The discovered gain-loss values are added to V_u . The termination condition, evaluated in line 8, tests the value added invariant for the use case being considered. If the invariant is satisfied, the procedure terminates; otherwise a new agent is identified by procedure `identify_new_agent`. This agent is found as follows. If the set of actors Act_u is non-empty, `identify_new_agent` removes one actor from Act_u and returns it. Otherwise, the procedure identifies a new counter-actor using the knowledge that for some value v

in V_u , $\sum_{j=0}^{|SM_u|} w_j v \neq 0$, where w_j is the weight associated with v in state machine $sm_j \in SM_u$.

```

procedure reduce_uc_to_sm( $u, Act_u,$ 
     $a : a \in Act_u, V$ )
1)  $A_u = \{a\}$ 
2)  $V_u = \emptyset$ 
3)  $Act_u = Act_u \setminus \{a\}$ 
4) loop
5)    $sm_a = \text{generate\_sm}(a, u)$ 
6)    $V_u = V_u \cup V_a$ 
7)    $SM_u = SM_u \cup sm_a$ 
8)   if  $\sum_{sm_i \in SM} w_i v_i = 0$  and  $Act_u = \emptyset$  then
        exit
9)   else
10)     $a = \text{identify\_new\_agent}()$ 
11)     $A_u = A_u \cup a$ 
12)  end if
13) end

```

Figure 13. Procedure `reduce_uc_to_sm`.

```

procedure reduce_sm_to_cld_oclspec ( $SM_u$ )
1) forall  $sm_a \in SM_u$  do begin
   // extend the CD
2) create class for agent a and
   stereotype it as such
3) forall transitions  $t \in sm_a$  do begin
    $v, w$  -- the value and weight
       associated with  $t$ 
4) if no class for  $v$  then
5) create value class for  $v$ 
6) end if
7) if  $w > 0$  then
8) create association link from
   the value class to the agent
   class
9)  $E_{post} = E_{post} \cup \{e_v\}$  -- write a
   postcondition involving the
   value class and the agent class
10) else
11) create association link from
   the agent class to the value
   class
12)  $E_{pre} = E_{pre} \cup \{e_v\}$  -- write a
   precondition involving the
   value class and the agent class
13) end if
14) add necessary collaboration paths
15) end
16) write service invariants from the
   agent under consideration point of
   view and update  $E_{inv}$ 
17) end

```

Figure 14. Procedure reduce_sm_to_cld_oclspec.

Procedure reduce_sm_to_cld_oclspec, shown in Figure 14, has m iterations, where $m = |SM_u|$ is the cardinality of the set of state machines passed as an input. The state machines are processed one at a time. In each iteration, the CD is extended with new classes and/or class relationships as follows. A class is created for the state machine's agent and stereotyped accordingly. A class (if one does not already exist) is created for each value gained or lost by the agent. If the value is lost, a unidirectional association link from the agent class to the value class is added. If the value is gained, the direction of the association link is opposite, i.e., from the value class to the agent class. For each new class, the procedure adds the relationships needed to create the collaboration paths (with classes other than the agent class) necessary for the use case execution. Analogous to CD, the use case OCL specification is extended in each iteration. The context of the new OCL constraint is an instance of the agent whose state machine model is being processed. A pre-/postcondition is defined for each value lost/gained by the agent on line 9/12. For each new constraint the procedure adds the expressions necessary to specify the semantic relationships with objects other than the instance

of the agent class under consideration. Finally, on line 16, invariants (frame assumptions) are defined if necessary.

The described process takes place in analysis. It, therefore, can and often does involve backtracking. Our initial experience shows that backtracking is less frequent and less deep than with any other approach for system decomposition and allocation of external interactions to component interactions, refer to Section 1.

5. Conclusion

In this paper we provide a link between use cases and class diagram. We proposed a method for deriving formal use case specification in OCL. The presented method is based on a conservation law called value added invariant. We used the algebraic invariant as a means of discovering classes, class relationships and OCL specifications from a narrative use case description. The use case specification is defined as a set of preconditions, postconditions and invariants over the business objects exchanged during the performance of a use case. All constraints are specified from an agent's, that is, partial point of view. The derived OCL expressions define declaratively the use case under consideration. An OCL parser can be used to check the correctness of the model. We developed an algorithmic procedure that transforms almost mechanically a set of use cases to class diagrams and OCL constraints. This procedure, based on the state machine model, fills the gap between the outside behavioral system description as offered by the use case model and the "first cut" at system architecture, the analysis level class model. The proposed approach resolves the vertical consistency problem between a use case and its use case realization-analysis. It reinforces Jacobson's most important factors of success, namely formalizing the use case specification and system development through model transformation. Another advantage is that we consistently stay at a high-level of abstraction since we work with business values exchanged in the course of the use case execution, thus defining a clear point where analysis ends and design begins. The right level of detail is no longer a matter of personal preference.

We are gathering statistical data on backtracking frequency and backtracking depth of the proposed algorithmic procedure. We are developing an algorithm for minimization of the state machine model preserving the classes and relationships resulting from the initial state machine. We are working on the integration of the proposed approach for use case specification in an experimental CASE tool for object-oriented design incorporating an OCL parser. We are developing a method for requirements verification through model animation, similar to the LTSA [19], that will use as an input the agent state machine descriptions, business objects and OCL constraints. Having declarative formal use case specifications opens up interesting research

topics such as automatic generation of runtime constraint checking implementations and automatic test generation. There is a natural relationship between the externally visible use case behavior and control classes in the BCE (MVC) approach. We plan to explore further the connection between use cases, value added invariants, OCL specifications and sequence diagrams.

References

- [1] S. Ambler, "Toward Executable UML," *Soft. Development*, Jan. 2002.
- [2] B. Baber, "Software: Engineering or Pre-engineering? or How (not) to blow up the Ariane 5 and DM1,200 million in 40 seconds," *Faculty of Science Lecture*, Wits University, Johannesburg, SA, May, 1999.
- [3] A. Bahrami, *Object Oriented Systems Development*, McGraw-Hill, 1999.
- [4] K. Bittner, "Why Use Cases are not Functions," *The Rational Edge*, Dec. 2000.
- [5] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language*, Addison-Wesley, 1999.
- [6] A. Borgida, J. Mylopoulos, and R. Reiter, "On the Frame Problem in Procedure Specification," *IEEE Trans. On Soft. Eng.*, 21(10), 1995.
- [7] K. Buhner, "From Craft to Science: Searching for First Principles of Software Development -- Part I," *The Rational Edge*, Dec. 2000, http://www.therationaledge.com/content/dec_00/f_craftscience.html.
- [8] A. Clark and A. Moreira, "Use of E-Lotos in adding formality to UML", *Journal of Universal Computer Science*, 6(11), 2000.
- [9] A. Cockburn, "Structuring use cases with goals," *Journal of Object-Oriented Programming*, Sep/Oct and Nov/Dec, 1997.
- [10] R. Collard, "Small Change, Big Trouble," *STQE* 4(1), Jan./Feb. 2001, p.14.
- [11] E. Dijkstra, "The End of Computing Science," Manuscript written for the CACM, Nov. 2000.
- [12] A. Ek, "Telelogic, the SOMT method," 1995, <http://www.telelogic.com/download/paper/somt.pdf>.
- [13] G. Engels, J. Kuster, R. Heckel, and L. Groenewegen, "A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models," *ACM ESEC/FSE*, Vienna, Austria, 2001.
- [14] R. France, "A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts," *Proc. of OOPSLA*, 34(10), Oct. 1999.
- [15] J. Ivari, "Object-Oriented as Structural, Functional and Behavioral Modeling: A Comparison of Six Methods for Object-Oriented Analysis," *Inf. and Software Technology*, 37(3), 1995.
- [16] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
- [17] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
- [18] L. Maciaszek, *Requirements Analysis and System Design*, Addison Wesley, 2001.
- [19] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, John Wiley & Sons Ltd., 1999.
- [20] R. Milner, *Communication and Concurrency*, International Series in Computer Science, Prentice-Hall, 1989.
- [21] A. Moreira and J. Araújo, "Generating object-Z specifications from use cases," *International Conference on Enterprise Information Systems*, Setúbal, Portugal, March 1999.
- [22] Object Management Group, *OMG Unified Modeling Language Specification, Version 1.4*, Sept. 2001, <http://www.omg.org>.
- [23] D.L. Parnas, "Software Engineering: An Unconsummated Marriage", *Communications of the ACM*, 40(9), September 1997, p. 128.
- [24] C.A. Petri, *Kommunikation mit Automaten, Schriften des Rheinisch, Westfalischen Institutes für Instrumentelle Mathematik and der Universität Bonn*, 1962.
- [25] B. Roussev, "The Value Added Invariant: A Newtonian Approach for Generating Class Diagrams from a Use Case Model," *WITUML, 16th European Conference on Object-Oriented Programming ECOOP 2002*, Malaga, Spain.
- [26] The Standish Group, <http://www.standishgroup.com>.
- [27] J. Warmer, "The Future of UML," *OMG Information Day*, Amsterdam, March, 2001.
- [28] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, Reading, MA, 1998.
- [29] R. Wieringa, "A Survey of Structured and Object-Oriented Software Specification Methods and Techniques," *ACM Computing Surveys*, 30(4), Dec. 1998.
- [30] R. Wirfs-Brock and B. Wilkerson, "Object-oriented design: A responsibility driven approach," *In Proc. of OOPSLA '89*, pp.71-75, 1989.