Transforming Use Case Models to Class Models and OCL-Specifications

Boris Roussev
Information Systems Dept.
University of the Virgin Islands
Box 10,000, Kingshill, VI 00850
brousse@uvi.edu

Jie Wu
Computer Science and Engineering Dept.
Florida Atlantic University
Boca Raton, Fl 33431, USA
jie@cse.fau.edu

Abstract: 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 business values exchanged in a use case guides the design of statechart descriptions for the actors and counter-actors, collectively called agents, of a use case. Each state in a statechart corresponds to a system state characterized by sending or receiving of a business object to or from the system's environment. The system class model and OCL specifications are derived from the agents' statecharts. 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.

Keywords: *software engineering, UML, object-oriented analysis, object-oriented modeling, model transformation, analysis techniques*.

1. Introduction

The contention of [11] and [27] supported by the empirical data in the CHAOS report [3] is that object-oriented methods are underdeveloped in the specification of external functions of systems and weak in guidelines for partitioning a system into components. The work of Jacobson [12] has proven to be one of the most significant advances in software engineering. Jacobson's approach to software development was embraced by OMG and evolved into UML [19]. In UML, a system model consists of several partial models, representing different aspects of the designed system, such as, structural, behavioral, communication or interaction. The success of UML is based on the realization that the semantic gap between the problem and machine domains cannot be bridged by a single model. As a result, software development is defined as model transformation from more abstract to more detailed and precise models.

Modeling from different viewpoints gives rise to vertical, horizontal, syntactic and semantic consistency problems [10]. However, the different models must be consistent for an implementation to be feasible. The objective of this work is to facilitate the development of consistent UML models, and in particular statecharts, class diagrams, and OCL declarative specifications from use cases. The main contributions of the paper enhance UML's major success factor, namely, development through model transformation.

In this work, we define a process for discovering classes, class relationships, and OCL constraints from a use case based on the notion of use case invariant introduced in [20]. The proposed approach fills the gap between the outside behavioral system description as offered by use cases and the "first cut" at system architecture, the analysis class model. We show how use cases, statecharts and class diagrams can seamlessly relate to each other.

The proposed techniques apply to the design of information systems with non-trivial user requirements and rich sets of usage scenarios such as e-commerce systems, including Web-based applications, controllers, and software systems implementing enterprise business processes. The proposed approach is not suitable for system software or scientific applications development as the complexity of these systems is dominated by the design of their algorithms rather than modeling their functional requirements.

In complex environments, systems development starts with business modeling. The business use case and business object models specify how the designed system interfaces with business units, customers, users, and existing enterprise systems, such as inventories and supply chains. These business-level models can be used as an input to the proposed requirements analysis method.

The rest of the paper is structured as follows. In Section 2, we present the value added invariant. In Section 3, we introduce a process generating class diagrams and OCL use case specifications from the use case model of a system. Next, in Section 4, we give the algorithm describing formally the proposed approach. Then, in Section 5, we present a case study, followed by a discussion. In the final section, we outline plans for future work and conclude.

2. The Value Added Invariant of a Use Case

2.1 Background

In the UML community, there is an overall agreement that system functions are rendered as use cases. A *use case* is a cluster of related usage scenarios, where each scenario is a sequence of

transactions performed by an actor in a dialogue with the system that brings value to the actor [12]. 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 stereotyped <<realize>>) with collaborations (use case realizations), use cases indirectly determine the structural part of the analysis model, i.e. the conceptual system decomposition. Therefore, use cases, directly or indirectly, express functional, behavioral, communication and structural system properties. Cockburn [4] observes the confusion resulting from the different perceptions of use cases, and gives 24 possible interpretations of the use case model.

Two types of use case description techniques can be considered—imperative and declarative [19]. The advantage of an imperative specification is that it leads to an executable specification, which 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, statecharts, and sequence diagrams, while declarative techniques are only mentioned as a possibility but not given in the UML specification [19].

Several methods have addressed the problem of declarative specifications in object-oriented modeling. Although not directly related to use cases, Fusion [5] was among the first methods that advocated the idea of specifying declaratively the services offered by a system to its environment. Influenced by the formal specification languages Z [23] and VDM [14], Fusion employs schemata written in structured natural language, thus, reducing the cost of introducing the method in practice. [4] adds goals to the use case model to disambiguate its semantics and streamline its description. Catalysis [22], a synergy of ideas from Fusion and Syntropy [6] rendered in UML, introduces OCL post-conditions over domain types to define declaratively the effect of a system service. ANZAC [21], building on both Catalysis and Fusion, splits the use case functional specification into two separate models to mitigate the conflicting agendas of end users/owners and developers.

In this work, we use OCL pre- and post-conditions over exchanged business objects to describe declaratively the effect of a use case. We also introduce a new imperative technique describing the behavior of a use case as a set of communicating state machines [16], and we use these behavioral descriptions to derive the system's analysis-level class model.

Several works provide insights into how scenario-based visual formalisms can be linked to object-oriented models [7],[24],[26]. Less attention has been given to how scenarios can be described without developing or revealing the underlying system structure. For example, in a UML sequence diagram, lifelines correspond to instances of already identified classes or subsystems, while locations correspond to object states. In contrast, we propose to specify use case scenarios with a formal and interactive model including only outwardly visible communication events and information objects.

There are five basic approaches to class discovery from a functional system specification: CRC [28], Noun-phrase, Common class pattern, Use case driven [12], and Mixed [15]. The basic disadvantage of all five methods is the lack of pragmatic guidelines that can steer the process of class discovery and serve as a 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 can lead to imbalance between iterative and incremental—thrashing. In contrast, the approach presented in this paper gives practical, quantitative guidelines for use case specification and class discovery.

OCL is a formal and pure expression language used to specify constraints in UML models [25]. OCL expressions augment graphical models, e.g., class diagrams, to produce unambiguous and precise system descriptions. Visual models define some constraints, like association multiplicity, but in OCL we can specify richer ones, such as uniqueness constraints, formulae, limits and business rules. OCL constraints improve precision and communication, and facilitate design by contract. According to Ambler [1], the disadvantages of OCL are its poor readability and its inefficiency in specifying requirements-level and analysis-level concepts. In this work, we show that OCL can be used productively in expressing analysis-level concepts.

2.2 Value Added Invariant

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 an existing object. The respondent interacting with the actor initiating the execution of the use case instance is called *counter-actor class*. Actors and counter-actors are called collectively *agents*. For each use case we define the following conservation law. The joint distributed count of business values exchanged between the actors and the counter-actors in the course of a use case execution is constant. We call this property *the value added invariant of a use case*.

Jacobson et al. [13] give two criteria helpful in determining the scope of a use case, the key phrases being “resulting value” and “participating actor.” The authors went a step further by saying that “In some cases, actors are willing to pay for the value returned,” but stopped short of generalizing and defining the exchange of values. Below, using an online reservation system as a running example, we show how the exchange of business objects in the course of the execution of a use case instance is defined by a value added invariant and how this invariant can be used to discover the logical system structure.

Book Ticket Use Case

Description:
This use case describes how customers with HTML 4.0 compatible browsers can book tickets online for a selected performance.

Actor(s):
Customer, Clerk, Bank

Flow of events

Basic Flow

1. The use case execution begins when a customer points their browser to the book ticket web site.
2. The customer selects a performance, and in response, the system displays a seating chart with seats available for the show.
3. The customer picks a free seat. In response, the system displays a form collecting payment information.
4. The customer provides the required payment information and submits the payment form to the system. The system, in cooperation with the bank, carries out the payment transaction and responds by prompting the customer to provide their shipment address.
5. The customer fills out the shipment address and submits the form to the system. The system notifies the clerk of the customer’s purchase. Then, the clerk ships the ticket and records the shipment with the system. At this point, the system sends the customer an electronic confirmation and an electronic ticket.
6. Finally, the customer receives the shipment with the ticket.

Exceptional Flow of Events:

- The customer can cancel the purchase at any point prior to submitting their shipment address.
- In step (3), no vacant seats are available. The use case terminates.
- In step (3), the customer has selected a booked seat. The system displays an updated seating chart.
- In step (4), payment data is incomplete or incorrect. The system requests the missing information. If the payment information is incomplete or incorrect again, the system aborts the booking and rolls back to its state prior to the current purchase.

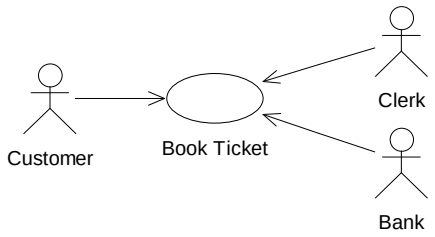
Pre-conditions

None

Post-condition

- A seat is booked.
- An amount equivalent to the seat's price is transferred from the customer's bank account to the system's bank account.
- The ticket for the booked seat is e-mailed to the customer.

(a) Use case description



(b) Use case diagram

Figure 1. `Book Ticket` use case.

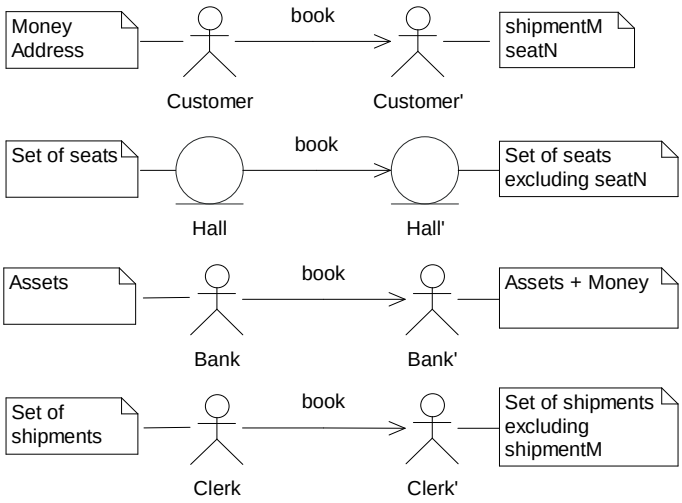


Figure 2. Value added invariant for use case `Book Ticket`.

Figure 1 shows a use case with three actors. Actor `Customer` initiates the use case execution. A customer books a seat and receives a ticket by providing in exchange some form of money. The respondent receiving the customer's payment is actor `Bank`. Actor `Clerk` is the respondent completing the bookings and shipping the ticket. From the use case model, it is not clear who provides the customer with a seat. We introduce a counter-actor, named Hall, providing a vacant seat. Prior to introducing Hall, the invariant for the use case was not satisfied be-cause the count of business objects received or sent by actor Customer did not match with the business objects gained or given away by actors Clerk and Bank.

At the highest level of abstraction, we define the evolution of an agent as a contract, which is an exact specification of the agent's interface. The business values sent/received (pre-/post-conditions) by the agent are described in the notes attached to the left/right agent snapshot, as it is shown in Figure 2. The agent snapshots and business values specify the state of the agent before

and after the execution of the use case. The invariant for `Book Ticket` expresses the fact that the joint distributed count of business objects before the execution of the use case is equal to the joint distributed count of business objects after the execution of the use case.

3. Deriving Class Diagrams and OCL Specification from a Use Case

3.1. Describing a Use Case as a Set of Statecharts

The lifecycle of each agent is described with a statechart. The statechart model can represent a system at an arbitrary level of detail. In analysis it is mandatory to stay at a high level of abstraction, i.e., out of design. This requirement is met by associating with each state a business object, which is received or sent by the action of the transition leading to that state. Figure 3 shows the statechart for `Customer`, derived from the main use case scenario. Since to each agent receiving a value corresponds an agent providing this value, we number the states so that they can be re-

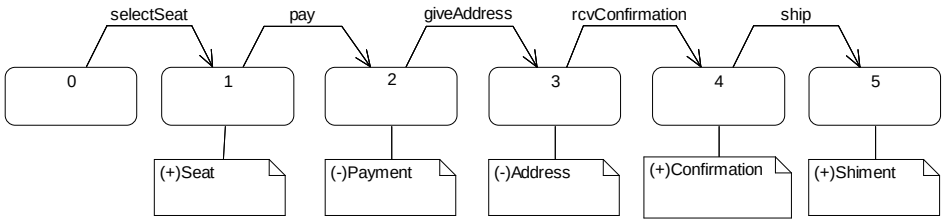


Figure 3. Statechart for `Customer`.

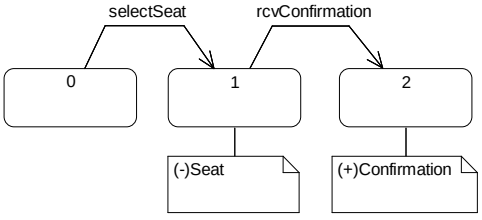


Figure 4. Statechart for `Hall`.

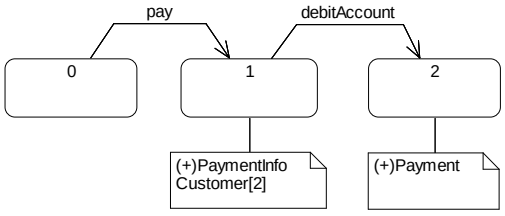
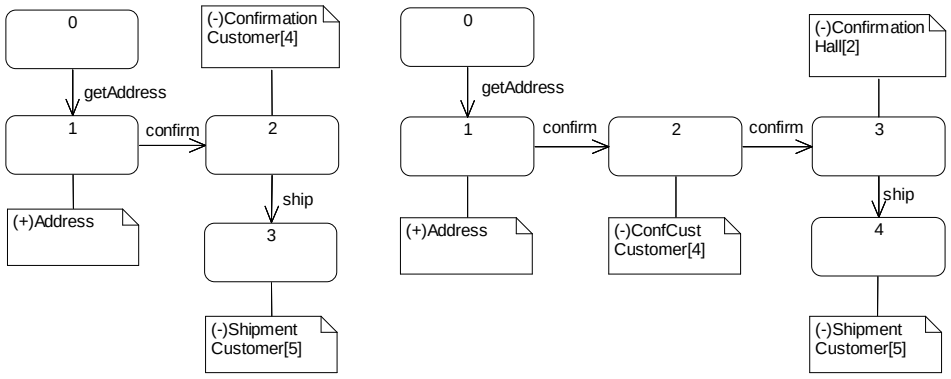


Figure 5. Statechart for `Bank`.



(a) Initial statechart **(b)** Modified statechart

Figure 6. Statecharts for Clerk.

ferred to unambiguously from other diagrams. A reference is written in the form of `agent[number]`, and it is placed in the note attached to the referring state, e.g. `Customer[2]`. There is a weight associated with each business object. The weight is +1, when the agent gains the business object, and -1, otherwise. For example, in Figure 3, `Customer` receives a `Confirmation` in the transition to state `Customer[4]`, and therefore, the weight for the `confirmation` object is +1. The weight magnitude can be omitted from diagrams. The statecharts for `Hall`, `Bank`, and `Clerk` are shown in Figures 4, 5 and 6(a), respectively.

In order for an invariant to be satisfied, the sum of the weighted business objects in the agent statecharts must be zero. The invariant is written as a set of equations. The equations represent in a testable way the information recorded in the statecharts. For example, the equations for objects `Seat` and `Confirmation` derived from the diagrams in Figures 3, 4, and 6(a) are written as:

$$(+1)\mathtt{Seat}_{\mathtt{Customer[1]}} + (-1)\mathtt{Seat}_{\mathtt{Hall[1]}} = 0 \qquad \text{(Eq.1)}$$

$$(+1)\mathtt{Confirmation}_{\mathtt{Customer[4]}} + (+1)\mathtt{Confirmation}_{\mathtt{Hall[2]}} + (-1)\mathtt{Confirmation}_{\mathtt{Clerk[2]}} \neq 0 \qquad \text{(Eq.2)}$$

The value added invariant is not satisfied in Equation 2. To balance the invariant, agent `Clerk` must send out two confirmations—one to `Customer` and another one to `Hall`. To deal with this, we add a new state and a transition to the statechart for `Clerk`, as it is shown in Figure 6(b).

Formally, the value added invariant for a use case can be expressed as $\sum_{i=1}^{k}\sum_{j=1}^{m} w_{ij}v_i = 0$, where $V_u = \{v_1, v_2,\ldots, v_k\}$ is the set of business values exchanged in use case u, $A_u = \{a_1, a_2,\ldots, a_m\}$ the

set of agents of u, and $W_u = \{w_{11}, w_{12,} \ldots, w_{1m}, w_{21}, w_{22,} \ldots, w_{2m}, \ldots, w_{k1}, w_{k2,} \ldots, w_{km}\}$ is the set of weights, with w_{ij} being the weight associated with business value v_i in agent a_j.

3.2. Reduction of a Set of Statecharts to a Class Diagram

Similarly to context diagrams, actors model the communication between the system and its environment. Maciaszek [15] observed an interesting dichotomy with regard to actors. On the one hand, actors are external to the system. On the other hand, actors are also internal because the system must maintain information about them so that it can knowingly interact with them. Hence, the specification needs to hold two models related to actors—a model of the actors and a model of what the system records about the actors.

We use the dichotomy of actors in a heuristic for discovering entity classes. The system must create an object of class `Customer` (if the object already exists it is linked) for each ticket booking. The agent statecharts are considered one at a time. The order of their processing is not significant. A statechart not processed yet is selected, and a class for its agent is created and stereotyped accordingly, see Figure 7. Then, a class for each value gained or given away by the agent is created. If the value is given away, a unidirectional association link is drawn from the agent class to the value class. If the value is gained, the direction of the association link is opposite. Finally, the developer determines and adds any associations necessary to create the collaboration paths between the discovered classes, e.g. association *sendTo* in Figure 7. Since the use case invariant does not account for these associations, the associations should be based on the business rules in the application domain.

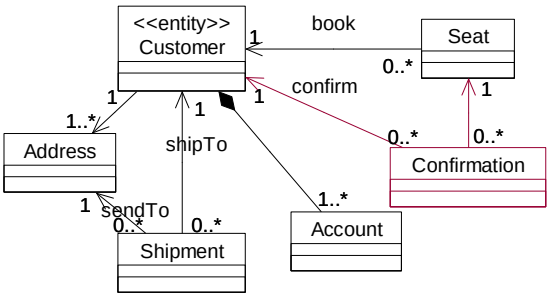


Figure 7. Class diagram generated from the statechart model of `Customer`.

The following is a commonly occurring pattern. An agent creates a new business object, e.g., a confirmation or a transaction ID, and then, provides it to another agent. The information about the exact timing of object instantiation is not represented in the use case invariant because the invariant is primarily concerned with the redistribution of business values between the system and

its environment and the direction of each redistribution. Over a sufficiently long period of time, the providing agent will create and deliver more than one object. We suppress the information about the exact moment of object creation by modeling the providing agent with a set of objects, e.g., a set of IDs, and by showing that after the execution of the use case instance, the cardinality of this set has decreased by one. In the general case, the object being sent has been created in the step preceding its sending. From the receiver's point of view, there is nothing unusual. As far as the receiving agent is concerned, it has got its business value.

Figure 8 shows how as a result of analyzing the statechart model for `Hall`, the initial class diagram has been extend with a new counter-actor class and two association links. Domain knowledge suggests the use of containment relationship between `Hall` and `Seat`. The class diagram after processing agent `Clerk` is presented in Figure 9. Agent classes are stereotyped as entity classes to record the long-lived changes in the redistribution of business values in the system. Since each agent interacts with other agents we duplicate the agent classes and the clones form a pool of candidates for controller classes in the BCE design pattern [12]. This is justified by the behavioral aspects captured by the agent statecharts and the existence of natural collaboration paths between the agents and the business objects. Very briefly, in the BCE design pattern, entity classes model long-lived information that survives a use case along with all behavior

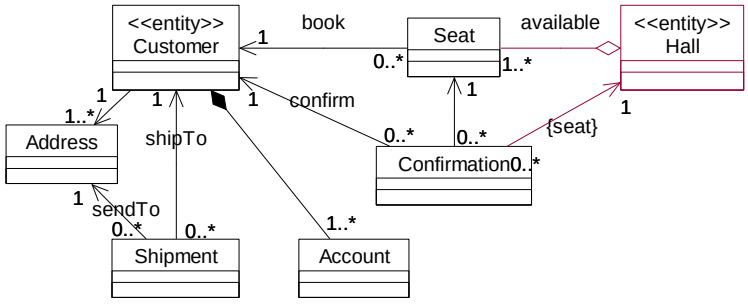


Figure 8. Diagram extended with knowledge from the statechart of `Hall`.

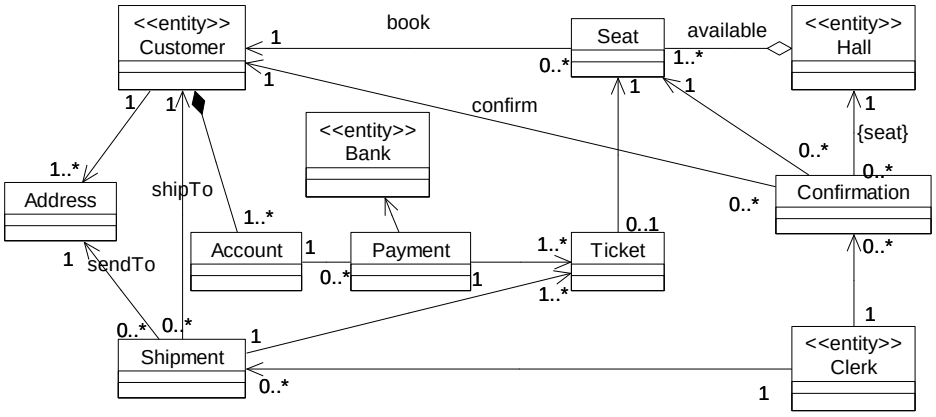


Figure 9. Final class diagram for `Book Ticket` use case.

naturally coupled to this information. Boundary classes model behavior and information dependent on the UI, and controller classes models functionality that is not naturally tied to any other class.

3.3 Converting a Set of Statecharts to an OCL Specification

This section presents a technique for deriving declarative use case specifications. We propose the use of OCL pre- and post-conditions over exchanged business objects to describe the effect of a use case scenario. The complete OCL specification for a use case is defined as the *exclusive-or* of the specifications for the individual use case scenarios.

Let us consider the statechart for agent `Customer`. We can define declaratively the behavior of the use case from the customer's point of view as a contract expressed in terms of OCL expressions. The contract is an exact specification of the service provided to the 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) post-conditions: a specification of the result, given that the preconditions are fulfilled. The service pre- and post-conditions are described in the notes attached to the agents on the left and right hand side of `Customer` and `Customer'`, respectively, as shown in Figure 10. We split complicated constraints into several separate constraints to improve their readability and writeability. The precondition for a customer is to have a valid credit card (called account for short) and a mail address that coincides with the account's billing address. The latter condition cannot be expressed in

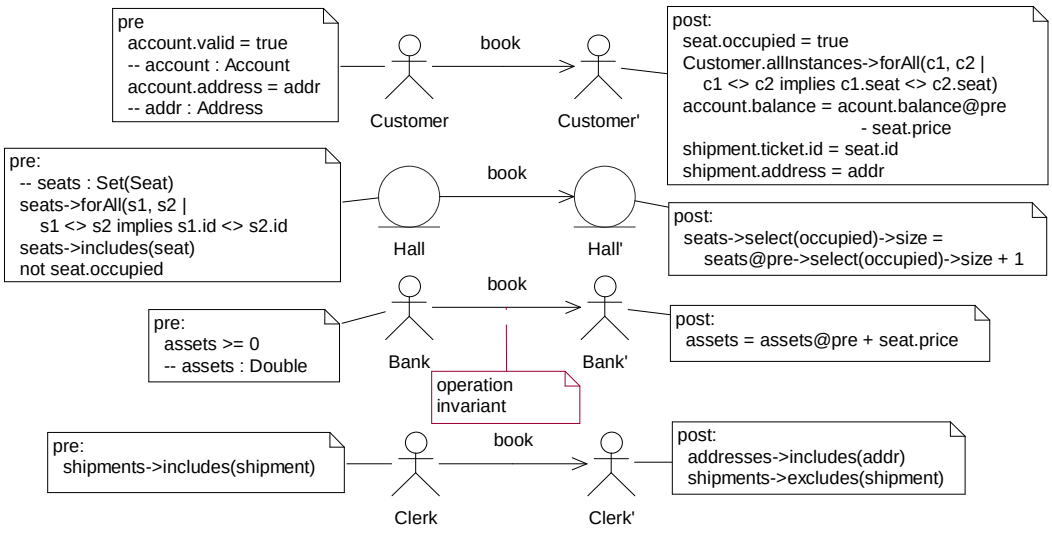


Figure 10. OCL specification for `Book Ticket` use case.

a graphical model. The post-condition includes an occupied seat, modified account balance and shipment. One of the post-condition constraints uses the *allInstances* predefined feature of OCL types. It is instrumental not only in specifying a uniqueness constraint, but also (implicitly through the *logic-and* of all post-conditions) in expressing 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 in the use case, see Figure 10. The OCL expressions are defined and evaluated in the context of the corresponding agent. *Set*(*Seat*) is used in a comment to clarify the type of collection *seats*. We use the extended variant (with two iterators) of *forAll* to define the uniqueness constraint on *seats*. *seats->includes(seat)* is a precondition indicating that *seat* has not been booked yet.

Declarative specifications may suffer from the so-called frame problem [2]. 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 = Seat.allInstances->select(seat | seat.occupied = true)
         ->iterate(s : Seat; result : Double = 0 | result + s.price)
```

In this case, we use *iterate* to add up the prices of all seats sold. When the *iterate* expression is

Notation	Meaning
U	Use case model
u	Use case
A	Actor or counter-actor
A_u	Agents in use case u
V	Gain-loss set of the model U
V_a	Gain-loss subset for agent a
V_u	Gain-loss subset for use case u
Act_u	Set of Actors for use case u
W	The set {-1,1}
sc_a	Statechart for agent a
SC_u	Set of statecharts 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.

evaluated, element *s* iterates over the collection of sold seats. The expression *result+s.price* is evaluated for each *s*. After each evaluation of the expression, its value is assigned to *result*.

4. Formal Process Description

This section describes unambiguously the method for use case specification and class discovery.

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

Definition 1. The statechart for an agent a, $a \in A_u$, is the ordered quadruplet $sc_a = (S_a, T_a \subseteq S_a \times S_a, V_a, gl_a)$, 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,

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

where $W = \{0,1\}$ is the set of weights. Function gl_a assigns to each transition t in T_a a value v from V_a weighted by w, $w \in W$.

The process of generating class diagrams and OCL specifications from a use case model is shown in Figure 11. The notation used is given in Table 1. The lower limit of the number of iterations in the process is determined by the number of use cases in use case model U. Procedure `reduce_uc_to_sc` 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 inputs and returns the set of statecharts SC_u for the agents in u.

```
      procedure generate_cld_oclspec_from_ucm(U: use case model)
1)    V = ∅
2)    forall usecases u∈U do begin
3)       loop
4)             outcome = reduce_uc_to_sc(u, Act_u, a, V)
5)             if outcome = success then
6)             exit
7)       end
8)       V = V ∪ V_u  -- add the identified business values to the
                      -- set of system business values
9)       reduce_sc_to_cld_oclspec(SC_u)
10)   end
```

Figure 11. Procedure `generate_cld_oclspec_from_ucm`.

The loop beginning on line 3 will continue to execute until the use case invariant becomes balanced. In line 8, the gain-loss set is updated with the new values discovered during the step of converting the use case to a set of statecharts. Procedure `reduce_sc_to_cld_oclspec` reduces the set of statecharts for the use case under consideration to one or more class diagrams and derives the use case OCL specification.

```
      procedure reduce_uc_to_sc(u, Act_u, a: a∈Act_u, V)
1)    A_u = {a}, V_u = ∅, Act_u = Act_u \ {a}
2)    loop
3)       sc_a = generate_sc(a, u)
4)       V_u = V_u ∪ V_a
5)       SC_u = SC_u ∪ sc_a
6)       if Σ_{j=1}^{m} w_ij v_i = 0 and Act_u = ∅ then
7)             return success
8)       else
9)             a = identify_new_agent()
10)            if a = null then
11)                  return failure
12)            else
13)                  A_u = A_u ∪ a
14)            end if
15)      end if
16)   end
```

Figure 12. Procedure `reduce_uc_to_sc`.

```
      procedure reduce_sc_to_cld_oclspec(SC_u)
1)    forall sc_a∈SC_u do begin
         // extend the CD
2)       create class for agent a and stereotype it as such
3)       forall transitions t∈sc_a do begin
            v, w -- the value and weight associated with t
4)          if class for v does not exist 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)             write a post-condition e_v involving the value
                     class and the agent class
10)            E_post = E_post ∪ {e_v}
11)         else
12)            create association link from the agent class
                     to the value class
13)            write a pre-condition e_v involving the value
                     class and the agent class
14)            E_pre = E_pre ∪ {e_v}
15)         end if
16)         add necessary collaboration paths
17)      end
18)      write frame assumptions from the point of view of the agent
            under consideration and update E_inv
19)   end
```

Figure 13. Procedure `reduce_sc_to_cld_oclspec`.

Procedure `reduce_uc_to_sc` is shown in Figure 12. 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_sc` takes an agent as an input and generates its statechart. The statechart design may entail changes to the existing statecharts such as adding new states and/or new transitions. The discovered gain-loss values V_a are added to V_u. The termination conditions, evaluated on lines 6 and 10, test if the use case invariant is balanced. If the invariant is satisfied and the set of actors Act_u is empty, the procedure terminates successfully; else 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 business value v_i in V_u, $\sum_{j=1}^{m} w_{ij} v_i \neq 0$, where $m = |SC_u|$ is the number of statecharts (one for each agent) and w_{ij} is the weight associated with v_i in statechart $sc_j \in SC_u$. If a new agent cannot be identified, procedure `reduce_uc_to_sc` returns failure, which causes backtracking in the top-level procedure. The order in which the agents are considered does not affect the resulting class diagram or OCL specification.

Procedure `reduce_sc_to_cld_oclspec`, shown in Figure 13, has m iterations, where $m = |SC_u|$ is the cardinality of the set of statecharts passed to it. The statecharts are processed one at a time. In each iteration, the class diagram and the OCL specification are extended with new classes/class relationships and OCL constraints, respectively. The context for the new constraints

is the agent whose statechart is being processed. A pre-condition/post-condition is defined for each value lost/gained by the agent on line 9/13. For each new constraint the developer adds the

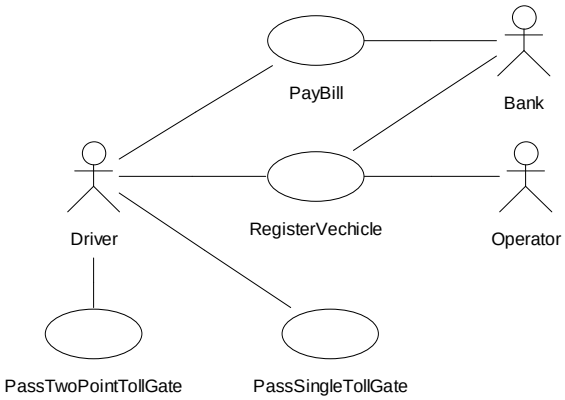


Figure 14. Use case diagram for the EZ pass system.

expressions necessary to specify the semantic relationships with objects other than the instance of the agent under consideration. If necessary, frame assumptions are added on line 18.

5. Applying the Process to a Distributed Software System

To test the feasibility of the proposed approach, we applied the invariant-based process to a real-world distributed system [17]. In the EZ Pass system, drivers of authorized vehicles are charged at tollgates automatically. They pass through special lanes called EZ lanes. To use the system, a driver has to register and to install an electronic tag (gizmo) in his/her vehicle. The vehicle registration includes the owner's personal data, account number and vehicle details. The owner's account is debited automatically at the end of every month. Each gizmo has a unique identifier that is read by the sensors installed at the tollgates. The information read is stored by the system and used to debit the respective account. The amount to be debited depends on the kind of the vehicle. When an authorized vehicle passes through an EZ lane, a green light comes on, and the amount to be debited is displayed. If an unauthorized vehicle passes through it, a yellow light comes on and a road camera takes a photo of the plate, used to fine the vehicle's owner (fine processing is outside the system scope). There are EZ lanes where the same type of vehicles pay a fixed amount, for example at a toll bridge, and EZ lanes where the amount depends on the type of vehicle and the distance traveled, for example on a highway. For the latter, the system stores the entrance tollgate and the exit tollgate.

The use case diagram for the EZ Pass system is shown in Figure 14. The textual descriptions for use cases `Register Vehicle` and `Pass Single Tollgate` are as follows.

Register Vehicle Use Case

Description:
This use case describes vehicle registration.

Actor(s):
Driver, Operator, and Bank

Flow of events

Basic Flow

1. A driver provides an operator with contact information, including their name and mailing address, the vehicle type and the vehicle's registration. In response, the system stores the information and prompts the driver to provide a valid bank account. This bank account which will be debited automatically at the end of each billing cycle.
2. The driver provides a bank account. The system verifies the account with the bank.
3. The system provides the driver with a gizmo and stores the bank account information, the gizmo ID associated with the registered car, and the starting date of the billing cycle.

Exceptional Flow of Events:

- The driver can cancel the registration at any point. The system rolls back to its state prior to the registration.
- In step (2), if the verification fails the registration is cancelled.

Pre-conditions

- The driver has a valid vehicle registration and a valid bank account.

Post-condition

- The driver receives a gizmo with a unique ID
- The driver, vehicle and gizmo are entered in the system.

Pass Single Tollgate Use Case

Description:
This use case describes the system's behavior in response to a vehicle passing through a single tollgate.

Actor(s):
Driver

Flow of events

Basic Flow

1. The use case begins when a vehicle with a gizmo passes through a single tollgate. The tollgate sensor reads the gizmo's ID. The system records the passage, including date, time, location, and rate, displays the amount the driver will be charged, and turns the green light on.

Exceptional Flow of Events:

- The gizmo is invalid or missing. The system turns the yellow light on and a photo of the vehicle is taken.

Pre-conditions

None

Post-condition

- The vehicle's account is updated with the passage information.

Figure 15 shows the value added invariant for use case `Register Vehicle`. The statecharts in Figure 16 are designed using procedure `reduce_uc_to_sc`. They verify that the joint gain-loss value sets for actors `Operator` and `Bank` balance with the value set for `Driver`. The class diagram and the OCL constraints generated from the set of statecharts using procedure `reduce_sc_to_cld_oclspec` are shown in Figure 17 and Figure 18, respectively. The class

attributes have been abstracted from the characteristic features of the business objects in the textual use case description. The post-condition on `Operator` involving the collection *registrations* and the keyword *@pre* is interpreted as providing a registration to a new vehicle.

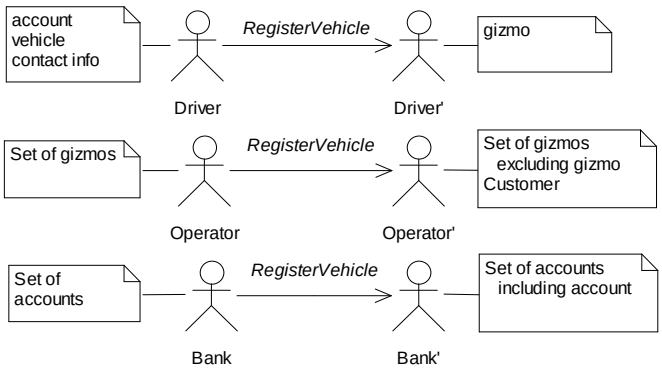


Figure 15. Value added invariant for `Register Vehicle` use case.

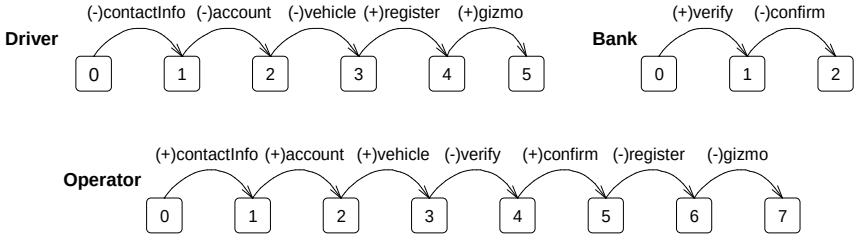


Figure 16. Agents' statecharts for the `Register Vehicle` use case.

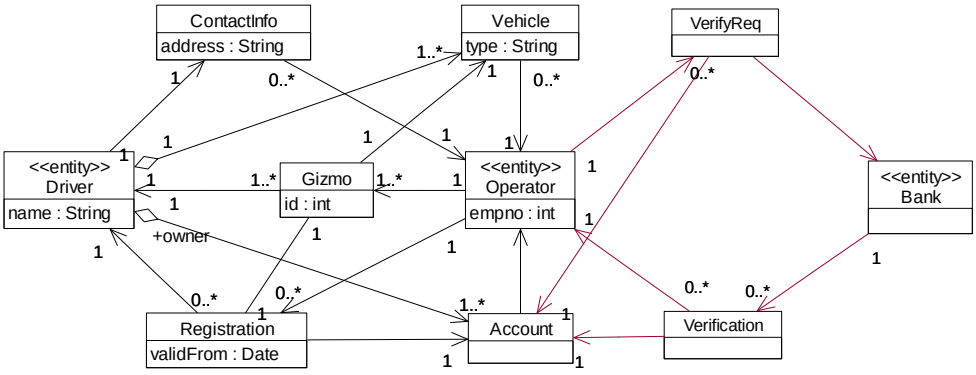


Figure 17. Class diagram for the `Register Vehicle` use case.

Next, we consider use case `Pass Single Tollgate`. The only actor interacting with this use case is `Driver`. The invariant is instrumental in identifying its counter-actors and in evolving further the system's software architecture. The driver exchanges electronic money (the fact that the actual payment occurs later is immaterial) for speedy passage. To model the creation of a passage object, we introduce a counter-actor, named `Counter` (meaning counting not counter-

acting), providing a passage every time a registered vehicle is detected at an EZ lane and calculating the amount of money due in exchange for that passage. To calculate the passage rate, agent `Counter` needs information about the passage's location. If agent `Counter` receives a location

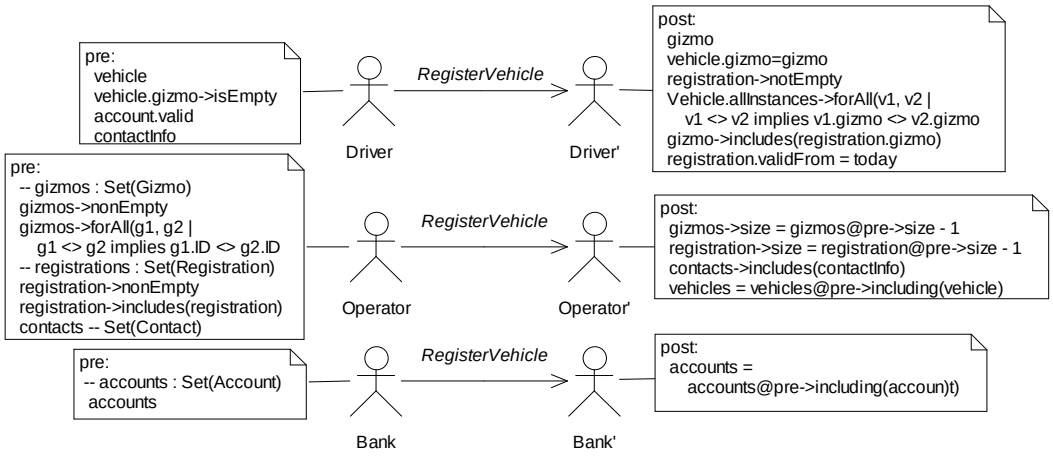


Figure 18. OCL specification for `Register Vehicle` use case.

value, then there must be an agent providing this value. To balance the invariant, we introduce a new counter-actor, named `Lane` in Figure 19, to provide the location value. The resulting class diagram and OCL specification are shown in Figures 21 and 22, respectively. Note that since `Gizmo ID` is related to `Registration`, which in turn is related through class `Gizmo` to `Vehicle` (see Figure 17), the vehicle type is known, and the amount charged can be determined based on the rate in `Price` and stored in `Passage`.

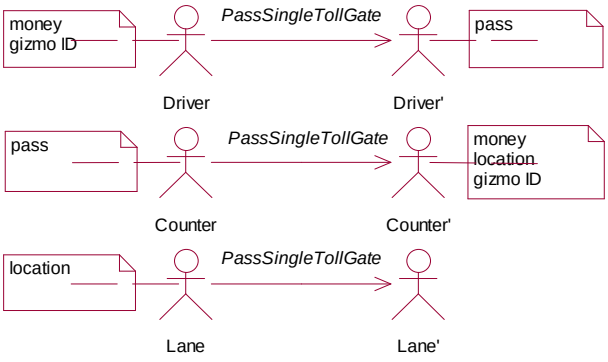


Figure 19. Value added invariant for `PassSingleTollGate` use case.

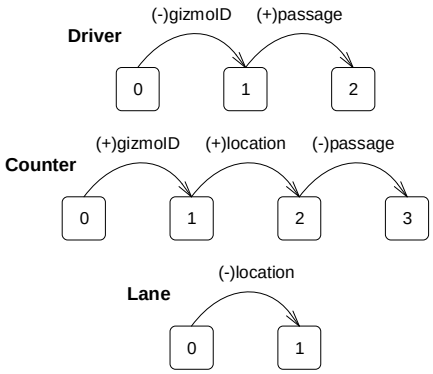


Figure 20. Agents' statecharts for the `PassSingleTollGate` use case.

The invariants, diagrams, and specifications for the other two use cases are designed similarly.

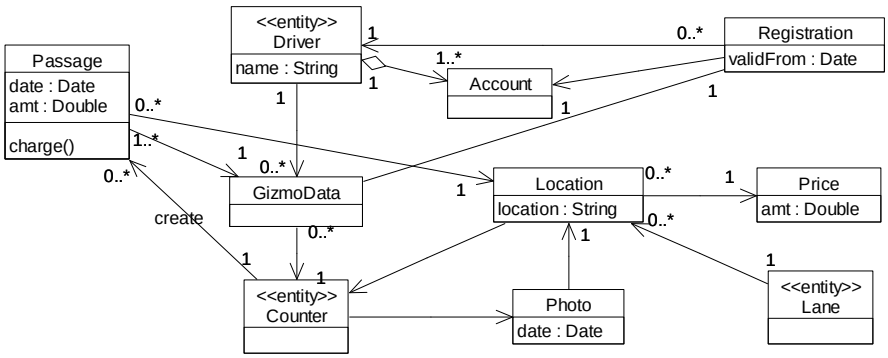


Figure 21. Class diagram for the `PassSingleTollGate` use case.

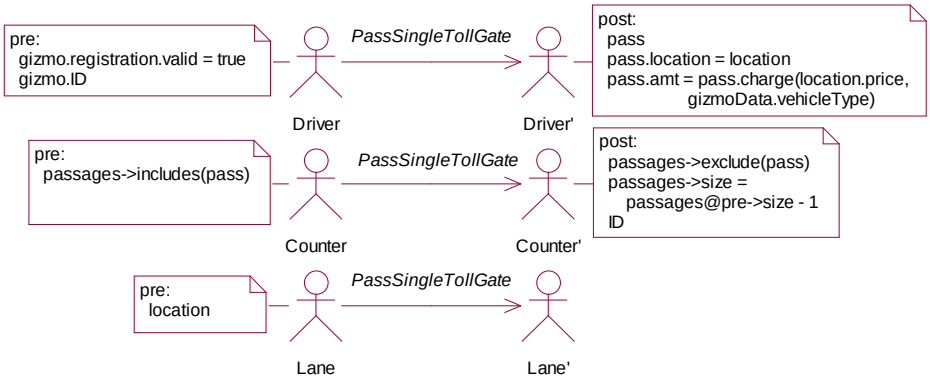


Figure 22. OCL specification for `PassSingleTollGate` use case.

EZ Pass is a mid-sized system of moderate complexity. Our experience with developing EZ Pass shows that the consistency of the designed models is better than that of the models produced with the CRC, Noun-phrase, Common class pattern, or Use case-driven approaches. We attribute this to the constraints imposed on the models' elements, e.g., classes, class associations, and association multiplicities, by the use case invariants and statechart diagrams.

6. Discussion

Our work is related to several object-oriented methods dealing with user requirements specification and analysis. We already discussed Jacobson's use cases. In what follows, we discuss how Fusion, Catalysis and ANZAC relate to the value added invariant.

In Fusion [5], a specification consists of an interface model and object model. The interface model includes declarative specifications of system operations, expressed as schemata written in natural language and the temporal ordering of actions captured by regular grammar expressions. The object model describes the information entities the system deals with in order to fulfill re-

quests coming from its environment. Our approach has a number of similarities to the analysis model in Fusion. In particular, analogously to Fusion operations, we define declaratively, but in OCL, the value brought to the actor initiating a use case execution. Instead of regular grammar expressions, we employ statecharts to define the logical order of the actions occurring on the system-environment border. Besides the differences between the two methods in terms of the formalisms used, schemata versus OCL expressions, regular expressions versus statecharts, and entity-relationship diagrams versus classes, there are deeper methodological differences. The value added invariant binds quantitatively the exchanged business objects. The statechart composition operation synthesizes the system structure from the behavioral descriptions of the use case agents. In Fusion, the object model makes a separation between the classes that lie within the system boundary and those ones that lie outside it. For us, this dichotomy is not absolute. We include the classes that lie outside the system boundary in the system model because in order to interact with them the system has to maintain knowledge about them.

Catalysis [22] is a component-oriented development method. Catalysis defines a use case as a goal-oriented collaboration (action) between a system and an actor. Catalysis uses OCL post-conditions over application types as a way of defining declaratively the effect of an action. Catalysis introduced the idea of representing change as instance snapshots diagrams expressing candidate types and their associations at a particular point in time. In these diagrams, the result of a use case execution is expressed by showing which associations are deleted and added to the diagram. The value added invariant takes the idea of snapshot diagrams a step further by expressing quantitatively the exchange of types (business objects in our parlance) and using the equations over types as a practical guideline for class discovery, test for completeness, and method of justifying types' existence. Catalysis focuses only on the outcome of a use case execution without formally describing the sequence of events leading to the outcome. Catalysis ignores the question of who would possess a type when the new owner is not an actor. To express the ownership change we define counter-actors. Our approach strengthens a major innovation of Catalysis—the type model of a use case as the link between a use case and the different UML diagrams, e.g., sequence and collaboration diagrams. An important distinction between Catalysis and the value added invariant is that Catalysis visualizes the type model, while we use the domain dictionary to compose the system architecture. Catalysis, influenced by SOMA [9] extends class descriptions with assertions and rules of the *if/then* and *when/then* form to encapsulate global knowledge in local entities. In its current form, the value added invariant relies on UML rules, e.g., OCL constraints on class models and statecharts. Both Catalysis and the value added invariant fall in the category of translational approaches. A translational methodology regards the development proc-

ess as a sequence of distinct models together with a procedure translating from one to the next, e.g., Executable UML. In the value added invariant the analysis level architecture is composed out of the agents' statecharts. We do not make use of the more subjective refinement.

ANZAC [21] is a methodology employing OCL in its declarative, goal-based use case specifications. ANZAC introduces new modeling artifacts, collectively called ANZAC specification, maintained separately from the use case specification. The latter adds extra cost to the software processes because of the need to maintain two different specifications. The extra cost should be offset by the benefits brought in by the new specification formalism. Adopting ANZAC would require staff training and process change, entailed by the introduction of the new modeling artifacts. In contrast, our goal is to enrich the analyst's arsenal of tools in an unobtrusive way.

In analysis, use cases are decomposed into conceptual components and their external interactions are mapped to component interactions. Cognitively, this includes the following activities: discovering components, discovering messages, allocating operations to components, and defining the components information structures. Since components support each other's information needs, the activities above are interdependent. The task of realizing a use case is, by and large, explorative, and it may involve backtracking. Guidelines for realizing a use case and for assessing the goodness of a use case realization other than drawing partially constructive sequence diagrams representing individual scenarios, are practically missing. Our work alleviates this situation because it provides a way to test analysis-level class models by checking if they satisfied the invariant of the use case they realize.

Harel refers to the stark difference between inter-object and intra-object behaviors as the grand duality of system behavior and argues that we are far from having a good algorithmic understanding of this duality [8]. In this respect, the invariant-based approach is a means for deriving the inter-object behavior, i.e., use case behavior, from the agents' intra-object behaviors, whose specification is guided by an algebraic invariant, steering the exploratory derivation process. Our experience shows, that the upfront investment in constructing the invariants is offset by reducing the probability of backtracking later in the development lifecycle. The proposed process of class discovery converges quickly because the invariants constitute unambiguous guidelines for developers to follow. The invariants narrow down the choices that developers have to make. Even though backtracking is possible, design changes beyond the initial stage of the use case realization are predominantly transformational.

There is a growing consensus that new techniques for formal modeling and for analyzing properties of the environment, as opposed to the behavior of the software, are needed [29], [18].

The demand comes from the realization that satisfactory analysis cannot be performed in isolation from the context in which the system will operate. The approach proposed in this work relates directly to context modeling, since it captures properties of actors and exchange of business domain objects crossing the system boundary.

We define precisely when a use case ceases to expect any more input. The value added invariant delimits the boundaries of a use case, thus serving as a pragmatic guideline to requirements engineers. The invariant addresses directly the problem of horizontal consistency. It ensures that the integration of new classes and relationships in the class model is functionally and semantically correct. The balance of the value added invariant serves as a litmus test for the quality and completeness of the software architecture. Since the invariant is defined over business objects, it safeguards system analysts from entering design (the infamous “analysis-paralysis” problem). The composition technique deriving class diagrams from agent statecharts resolves a major weakness of the UML languages for system's dynamics—the lack of seamless integration between state-change models and static structure models. The Unified Process (UP) and agile methods could benefit from incorporating the value added invariant as it guides the transition from informal requirements towards formal models. The discontinuity in model transformation observed in UP is diminished since the proposed invariant formalizes requirements modeling and environment modeling.

Conclusion

In this paper we proposed a method for deriving collaborations of classes realizing a use case model through a sequence of model transformations and for specifying formally use cases with OCL constraints. The presented method is based on the notion of value added invariant of a use case. We used these algebraic invariants as a means of discovering of classes, class relationships, and OCL specifications from narrative use case descriptions. The use case specification is defined as a set of pre- and post-conditions defined over the business objects exchanged during the execution of a use case instance. 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. We defined formally the proposed process, and demonstrated with a real-world system how it can be used by system analysts to transform a set of use cases to class diagrams and OCL specifications. This procedure fills the gap between the outside behavioral system description as offered by the use case model and 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 Jacob-

son's most important factor of success, namely, system development through model transformation. Currently, we are developing a method for requirements verification through model animation using LTSA [16] that will use as an input the agent statechart descriptions and business objects. Having declarative formal use case specifications opens up interesting research topics such as automatic generation of runtime constraint checking implementations and test generation (test cases and test procedures), which we plan to explore.

References

[1] S.Ambler, "Toward Executable UML," Software Development, Jan. 2002.

[2] A.Borgida, J.Mylopoulos, and R.Reiter, "On the Frame Problem in Procedure Specification," *IEEE Trans. On Soft. Eng.*, *21*(10), 1995.

[3] CHAOS, CHAOS Research Report, http://www.standishgroup.com, 2003.

[4] A.Cockburn, "Structuring use cases with goals," *Journal of Object-Oriented Programming*, Sep/Oct 1997.

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

[6] S.Cook and J.Daniels, *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Prentice-Hall, 1994.

[7] W.Damm and D.Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, *19*(1), 2001.

[8] B.P.Douglas, *Real Time UML*, 3rd Ed., Addison-Wesley, Boston, 2004.

[9] I.Graham, "Business Process Re-engineering with SOMA," *Proc. of Object Expo Europe*, Dorking, England, 1994.

[10] 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.

[11] J.Ivari, "Object-Orientation as Structural, Functional and Behavioral Modeling: A Comparison of Six Methods for Object-Oriented Analysis," *Inf. and Software Technology*, *37*(3), 1995.

[12] I.Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[13] I.Jacobson, G.Booch, and J.Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 99. 1999.

[14] C.Jones, *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

[15] L.Maciaszek, *Requirements Analysis and System Design*, Addison Wesley, 2001.

[16] J.Magee and J.Kramer, *Concurrency: State Models and Java Programs*, John Wiley & Sons Ltd., 1999.

[17] A.Moreira and J.Araújo, "Generating object-Z specifications from use cases," *International Conference on Enterprise Information Systems*, Setúbal, Portugal, March 1999.

[18] B.A.Nuseibeh and S.M.Easterbrook, "Requirements Engineering: A Roadmap," In A.C.W. Finkelstein (ed) *The Future of Software Engineering*, IEEE Computer Society Press, 2000.

[19] OMG, *OMG Unified Modeling Language Specification*, Version 1.5, March 2003.

[20] B.Roussev, "Generating OCL Specifications and Class diagrams from Use Cases: A Newtonian Approach," *In Proc. 36th Hawaii Int'l Conference on System and Sciences*, HICSS'36, Hawaii, 2003.

[21] S.Sendall, *Specifying Reactive System Behavior*. Ph.D. Thesis, EPFL, Lausanne, 2002.

[22] D.D'Souza and A.Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley 1998.

[23] J.M.Spivey, The Z Notation: A Reference Manual. Prentice-Hall, 1989.

[24] S.Uchitel, J.Kramer, and J.Magee, “Incremental Elaboration of Scenario-Based Specifications and Behavior Models Using Implied Scenarios,” *ACM Trans. on Software Engineering and Methodology*, *13*(1), pp 37-85, 2004.

[25] J.Warmer and A.Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, Reading, MA, 1998.

[26] J.Whittle and J.Schumann, “Generating Statechart Designs from Scenarios,” In Proc. of *22nd Int'l. Conference on Software Engineering*, ICSE’00, Limerick, Ireland, 2000.

[27] R.Wieringa, “A Survey of Structured and Object-Oriented Software Specification Methods and Techniques,” *ACM Computing Surveys*, *30*(4), Dec. 1998.

[28] R.Wirfs-Brock and B.Wilkerson, “Object-oriented design: A responsibility driven approach,” *In Proc. of OOPSLA’89*, pp.71-75, 1989.

[29] P.Zave and M.Jackson, “Four dark corners of requirements engineering,” *ACM Transcations on Software Engineering and Methodology*, *6*(1), 1-30, Jan. 1997.