A Lightweight Approach to Consistency of Scenarios and Class Models

Martin Glinz
Institut für Informatik, Universität Zürich
Winterthurerstrasse 190
CH-8057 Zurich, Switzerland
+41-1-63 54570
glinz@ifi.unizh.ch

Abstract

Today, object-oriented requirements specifications typically combine a scenario (or use case) model and a class model for expressing functional requirements. With any such combination, the problem of consistency between these two models arises.

In this paper, we present a lightweight approach to consistency between a scenario model and a class model. We assume semi-formal, loosely coupled models that are complementary: scenarios model the external system behavior; the class model specifies the internal, state-dependent functionality that cannot be expressed easily in a scenario (but is required to specify external behavior properly). We achieve consistency by minimizing overlap between the two models and by systematically cross-referencing corresponding information. We give a set of rules that can be used both for developing a consistent specification and for checking the consistency of a completed specification. Some rules can be checked automatically, the others are rules for manual inspection.

Keywords

Consistency, scenarios, class models, object-oriented requirements specification

1 Introduction

The initial approaches to object-oriented requirements specification were all centered around a class or object model, for example those of Booch [3], Coad and Yourdon [6] and Rumbaugh [25]. A short time later, scenarios and use cases emerged as an important means of requirements specification [5], [9], [16], [19]. Today, most approaches use a combination of structure, behavior and interaction models for (functional) requirements specification. Typically, structure and behavior are represented in class models that consist of a combination of class diagrams and statecharts, whereas interaction is modeled with use cases / scenarios [10], [26]. Only the Rational Unified Process [22] tries to model functional requirements with a use case model alone.

As soon as more than one model is used, the problem of inter-model consistency arises: how can we ensure that information in these models is neither contradictory nor partially incomplete? (A partial incompleteness is a situation where information given in one model requires corresponding information in another model which, however, is missing from that model.)

Nearly all requirements modeling techniques that use more than one model have *no systematic approach to combining the models consistently*. The consistency problem is simply ignored (and thus left to the requirements engineers and to the users who have to validate a requirements specification).

In this paper, we investigate the consistency problem between a scenario model (or use case model) and a class model. (Principally, we prefer object models to class models, because class models have some serious drawbacks, in particular where decomposability is concerned [13], [20]. However, in order to make our approach directly applicable in today's practice, we use class models here.)

Our main contribution is a concept for making a scenario model and a class model consistent. We use a lightweight approach for loosely coupled, UML-style models. We do not consider the problem of intra-model consistency here. That means, we assume that proper methods for establishing internal consistency for both the scenario model and the class model are in place. Our approach is lightweight in the sense that we use semi-formal models where consistency cannot be established formally. Instead, we systematically identify information that corresponds or is redundant in the two models, thus making manual inspection for consistency simple and straightforward. Some syntactic issues can even be checked automatically.

We are aware of only two other approaches to a systematic combination of scenarios / use cases and class models. Kösters, Pagel and Winter [21] couple use cases and class models by deriving the class model from the use cases and by annotating a graphic use case specification with the names of the classes that implement the use case. This is a design-oriented view where one proceeds from the use case model to a class model and from the class model to implementation. Buhr [4] regards scenarios as causal paths winding through a hierarchical object model.

Responsibility points bind segments of a path to elements of the object model. Object model and scenario paths are visualized together in so-called use case maps. Buhr's approach is design-oriented, too. Furthermore, the path through the object model is all we know about a scenario; there is no standalone specification of scenarios. Thus, the user-oriented separation of concern, which is one of the strengths of a scenario-based specification, is lost.

While we regard scenarios and a class model as complementary models that together form a requirements specification, neither of the approaches mentioned above does. Kösters, Pagel and Winter focus on the transition from scenarios to a class model. Buhr uses scenarios for visualizing the dynamics of the object model.

Other work on *detecting* inconsistency in requirements specifications typically deals with analyses based on a formal specification, for example [15].

Another track of related work deals with *managing* inconsistency in requirements specifications, which means living with inconsistency. These approaches typically require a formal specification too, and focus on managing intra-model inconsistencies that arise among different stakeholder viewpoints or within a single viewpoint [8], [17], [23].

The work by Grundy, Hosking and Mugridge [14] addresses inter-model consistency, which is more closely related to our problem. However, their approach differs fundamentally from ours. They focus on software development environment tools, managing inconsistencies in multiple views that are derived from a common repository. Artifacts in the repository are formally represented by a special kind of graph, thus allowing automatic inconsistency detection and management.

The rest of this paper is organized as follows. Chapter two starts with a short introduction to scenarios for requirements specification. Then we demonstrate when and why a purely external specification with scenarios is insufficient and must be augmented with a model of structure and state, typically expressed with a class model. This combination of different models leads to consistency problems that are characterized in Section 2.3. In Chapter three we present our lightweight approach to solving (or at least alleviating) the consistency problem. Chapter four presents an extensive example. In Chapter five we discuss rules for checking the consistency of models. We conclude with a discussion of achievements, open problems and future directions.

2 Modeling functional requirements with scenarios and class models

2.1 Scenarios

As the term "scenario" is used with several different meanings, we start with a definition.

DEFINITION. A *scenario* is an ordered set of interactions between partners, usually between a system and a set of actors external to the system. It may comprise a concrete sequence of interaction steps (instance scenario) or a set of possible interaction steps (type scenario).

Jacobson [19] coined the term *use case* for type scenarios and later introduced it into UML. Hence, this term is widely used today. In Jacobson's terminology, scenarios are always on the instance level.

Throughout this paper, we will use the term scenario. Where the distinction between type and instance level matters, we speak of type scenarios and instance scenarios respectively.

Scenarios have received considerable attention in requirements engineering for the following reasons.

- They describe the externally visible behavior of a system only, thus avoiding solution bias and premature design.
- They describe how users will work with a system, hence they are much easier to validate with users than for example class models or dataflow diagrams are.
- They can be used both for elicitation and description of requirements.
- They inherit the comfort and ease of natural language specifications, but avoid many of the problems of a purely narrative specification.

Typically, scenarios are used for the specification of functional requirements only. Some non-functional attributes (for example, performance requirements) can be included in scenarios. However, this is seldom done in practice. Other attributes (portability, for example) principally cannot be expressed with scenarios.

2.2 Why scenarios are not enough

A functional requirements specification can be regarded as a structured set of responses that are required as reactions to given stimuli. A principal means of capturing and representing this structure is to determine groups of stimuli and responses such that every group represents a system transaction or a system function from a user's perspective. A scenario is an excellent means of describing such a group of interactions. The particular strength of scenarios lies in the fact that they provide a decomposition of a system into functions from a user's perspective and that each such function can be treated separately – a classical application of the principle of separation of concern.

However, specifying the order of stimuli and responses for every single scenario does not capture the complete structure of stimuli and responses that make up a functional specification.

In addition to the internal structure of every scenario, we must be able to model the following:

- inter-scenario relationships,
- responses that depend on system state, i.e. on previous stimuli and their responses,
- an initial state (or an initial sequence of stimuli that produce this state).

Inter-scenario relationships can be included in scenario models by modeling them explicitly (e.g. Jacobson's

"uses" and "extends" relationships [19] or by expressing them with preconditions both for complete scenarios and for particular parts of a scenario.

Theoretically, state-dependent responses and the initial state of a system could be included in a scenario model, too. We would need to state pre- and postconditions for all state-dependent scenarios and use state variables in these conditions. Static relationships between state variables would have to be modeled by creating links between these variables in some postcondition. However, such a specification would become extremely clumsy for all systems with more than a few state variables. As every system that needs a database belongs to this category, this approach provides no practical solution for the specification of state-dependent requirements. As far as we know, nobody has ever tried to integrate state-dependent behavior specifications in this way in a scenario model. The Rational Unified Process [22], which tries to model functional requirements with scenarios only, says nothing about specifying state-dependent behavior and static structure. Consequently, this information will either be omitted in practice (thus leaving the specification considerably incomplete), or the business object model from the business modeling workflow will be used.

As a consequence, nearly all practical approaches that make use of scenarios combine the scenarios with a class or an object model. The scenarios capture functional requirements by specifying the behavior of a system as observed from a user's perspective. The class or object model, on the other hand, models both the static state space and the operations that modify it. (One could argue that models of a system's states should not be part of the requirements, because states are part of the system and thus should be regarded as design artifacts [18], [22]. This would be true if we modeled all the state variables required for the implementation of a system. However, if we only model those states that exist in reality as a part of a system's environment, but must be remembered and maintained by the system in order to perform its required tasks, we can regard these states as a genuine part of the requirements.

It should be noted here that we view scenarios and class model as being *complementary* to one another. Scenarios model the *externally observable, dynamic behavior* of a system, whereas the class model[1] specifies the *internal structure and behavior*, viz the data and operations that are required for determining the external behavior for state-dependent systems. This contrasts with other, more design-oriented views that regard the class model to be the implementation of the scenarios. The Rational Unified Process is a prominent example of the latter view.

2.3 The consistency problem

As soon as we augment a scenario model with a class model, the consistency problem arises: How can we ensure that the information in the scenario model is consistent with the information given in the class model?

We do not require formal models for scenarios and classes, so we cannot define consistency in formal terms. Our notion of consistency between models is that information given in the scenario model should *match* information given in the class model in every case where the two models interact or share information. Hence, we define inter-model consistency as follows.

DEFINITION. A scenario model and a class model are considered to be *consistent* if

(1) There are no contradictions between information in the scenario model and information in the class model (both where information is shared and where the models interact)
(2) In both models, there is no partial incompleteness with regard to the other model.

A partial incompleteness is a situation where information being present in one model requires corresponding information in the other model which, however, is missing from that model.

When viewing scenarios and class model as being *complementary* to one another, the following four situations require interaction between the scenario model and the class model and, hence, must be considered when developing an approach to inter-model consistency.

- Stimuli in a scenario that are not immediately required to produce a response will typically produce a state change which requires a corresponding operation or state transition in the class model.
- A response in a scenario that needs more data than the stimuli of this scenario provide requires stored data. This data, together with suitable access operations, must be represented in the class model.
- Every operation in the class model which is not referenced by another operation in the class model is typically used by a scenario in order to process a stimulus or to produce a response.
- Scenarios and a class model sometimes overlap. In particular, the scenario modeler is frequently tempted to include operations and data in a scenario that can also be found in the class model.

Gaps in a specification (situations where some information is completely missing because every model assumes that this information is contained in one of the other models) are considered to be a completeness issue, not a consistency one. However, when analyzing a specification for consistency, we frequently detect gaps, too. So gap detection can be regarded as a positive side effect of consistency checking (see Chapter 4 for an example).

2.4 Model collections vs. integrated models

A model that employs more than one modeling technique can be realized either as a collection of different models or as an integrated model with different views.

[1] including state models for the objects of the classes

In a *model collection*, every model in the collection is treated (created, maintained, used) separately. This makes life easy for language and tool designers (of course) and (apparently) also for the user. However, the ease comes at the expense of model quality, because inter-model consistency is hard to achieve for model collections:

- Models do not fit perfectly. There are typically conceptual overlaps between models on the one hand and gaps on the other hand.
- Inter-model consistency rules are often weak or not defined at all.
- Even where rules exist, they are frequently not supported by the available tools.

Thus, when a user wants to have inter-model consistency, working with model collections can be quite cumbersome.

An *integrated model* provides a single conceptual framework in which all model information is stored. Particular aspects (a scenario, an object structure, etc.) are generated from this framework by producing views. Thus, working with an integrated model is comfortable for the user. Through the view mechanism, she or he can still work on an isolated aspect, e.g. on a scenario model. By definition, there is no conceptual overlap in the model (views may overlap, but they are generated from a single common basis). Consistency rules are an integral, built-in part of the model.

Examples. Nearly all object-oriented modeling approaches are model collections; UML being the most prominent representative. Modern Structured Analysis [27] is another classic example. Information Engineering [24] is a variant of Structured Analysis which is based on the idea of an integrated model. In the field of object-oriented requirements specification we are developing a language and a tool called ADORA. ADORA is built upon the idea of an integrated model, using a hierarchical object structure as its basis [1], [13].

3 A systematic approach to consistency

3.1 Basic considerations

The achievable degree of consistency strongly depends on the degree of precision and formality that the scenario and class models have. For example, a scenario written in natural language cannot be formally made consistent with a class definition that also has informally defined attributes and operations. On the other hand, if we have formal models of both scenarios and class models, we can, to a large extent, formally interlink the models by formalizing consistency rules and building them into the metamodel.

In this paper, we pursue a *lightweight approach* to consistency that is intended to work with existing modeling techniques, helping to improve the consistency of requirements models written in UML, for example. Hence, we assume the scenarios and the class model to be loosely coupled (forming a model collection) and requiring only few formal elements. As we are working with semi-formal models, we cannot formally define and assess consistency. Our main idea is to establish consistency by systematic manual inspection and to support and simplify this process by

- minimizing overlaps between the models,
- systematically cross-referencing information between the models,
- defining a set of model construction and checking rules.

In Section 6.3 we briefly sketch how a *stronger approach*, based on an integrated model with formal syntax and partially formalized semantics could look.

3.2 Minimizing overlaps between the models

Overlap between a scenario and a class definition occurs when data and/or operations are modeled both in a scenario and in the definition of a class, or when some user interaction is included in the specification of an operation or of a state transition in the class model. The first situation typically occurs when a modeler requires persistent data for the specification of a required response. Overlaps of the second kind typically happen when a design-oriented person, who is unaware of the purpose of a scenario model, develops the class model.

Overlaps introduce redundancy into the specification. Redundancy always increases the probability of contradictions, in particular when the redundant information is located in two separate parts of the specification. Minimizing overlap therefore removes potential sources of consistency problems. However, accidentally overdoing overlap removal results in specification gaps. Hence, we formulate the following rules.

RULE 1 (Minimizing overlaps). (a) Do not re-model any data from the class model in a scenario, in particular, do not model any persistent data in a scenario. Model references to the specification of this data in the class model instead. (b) Do not model any user interaction in the specification of operations or state transitions in the class model. (c) Do not specify the details of an operation in both a scenario and a class definition. If the operation manipulates data specified in a class, model the operation within that class. Otherwise, model it in a scenario.

RULE 2 (Avoiding gaps). When avoiding or removing overlaps, never simply omit information from a scenario because you believe that it should be part of the class model and vice-versa. Instead, make sure that the information is present in the other model and include a reference to this information in your model.

3.3 Systematic cross-referencing

As our approach does not assume formal models for the scenarios and the class model, we have no formal way of determining information in the two models that corresponds and, hence, has to be consistent. Therefore, a cornerstone of our lightweight approach is to identify corresponding information explicitly by *cross-references*.

We first define the kinds of items that can be referenced and the format of references (below). Then we introduce two cross-referencing schemes; a simple scheme in Section 3.4 and an elaborate one in Section 3.5. An extensive example follows in Chapter 4.

Our cross-referencing concept is based on the following assumptions that are typical for today's object-oriented approaches to requirements modeling. The class model consists of class and state diagrams and of textual class definitions for every class. The syntax of the class definitions is defined so that we can formally identify names of classes, attributes, operations, etc. Scenarios are specified in a scenario or use case model. This model consists of an overview diagram and a definition of every scenario with structured text. The two models are loosely coupled, that means, they are part of a model collection. The UML [26] is a typical example for this kind of modeling.

A cross-referencing scheme makes little sense if it cannot be maintained and checked with tools. Hence we must be able to identify references formally in both kinds of models.

First, we determine the items that *can be referenced*. For the class model, this is straightforward: references to class names and to names of attributes and operations defined in classes must be possible. The same is true for state names in statecharts and for names of events and actions in state transitions. If a class model provides further formal elements, for example attribute domains or parameters and types for operations, then references to these elements shall also be possible. In the scenario model, we must be able to refer to the names of the scenarios. If we employ a step-structured textual description for the contents of a scenario (cf. Figure 1), references to step numbers should also be possible.

Second, we define the format of *references*. Within texts, we use an up-arrow as a tag symbol to identify a reference. For example, "Depending on ↑OrderProcessing.Choice the system displays the 'order entry' dialog or the 'browse' dialog" is a text in a scenario with a reference to a name Choice (presumably an attribute) in class OrderProcessing. In diagrams, we use directed dashed lines from the referencing item to the referenced one.

3.4 The simple scheme of cross-referencing

The simple scheme leaves scenarios and class model mostly as they are. Wherever in a scenario the creation of a response requires activity in an object of some class, we include a reference to the appropriate element in the class model (cf. Figure 4). The class model itself is not modified. This apparently primitive scheme makes sense for the following reasons:

- Scenarios are the primary means for users to understand a requirements specification. The class (or object) model augments the scenario model by specifying the state-dependent elements of the responses modeled in the scenarios. Thus references from the specification of an interaction in a scenario to the elements of the class model that are needed to understand this interaction are (a) helpful and (b) more important than references in the reverse direction.
- Simple referencing provides only little means for formal consistency checking. However, it provides considerable support for humans inspecting a specification for consistency.
- The effort required to include these references is very small in comparison with the benefit gained. In particular, referencing requires much less effort than drawing interaction diagrams for the same purpose.

3.5 The elaborate scheme of cross-referencing

In the elaborate scheme we specify the scenarios more formally, trying to express the scenario actions in terms of references to operations and state changes in the class model (cf. Figure 5). In this scheme, we also introduce references into the class model. We do so by defining stereotypes [2], [26] for the references. A stereotype «scenario» for classes allows us to include scenarios as boxes in a class diagram. A stereotype «uses» : list of names for dependency links models a usage reference from a scenario or class (at the tail of the dependency link arrow) to the items given in list of names in another class (at the arrowhead). Usage typically means that an attribute is accessed or an operation invocated. A stereotype «sends» : list of names models referencing by sending events or signals. Finally, stereotypes «creates» and «deletes» may be used to model creation and deletion references (cf. Figure 6). In order not to overload the class diagram, we may draw several diagrams, each one consisting of one scenario together with all classes referenced by this scenario or by each other.

The elaborate scheme requires considerably more effort than the simple one. But it also provides much more precision and details and allows for more formal consistency checking. So this scheme is particularly appropriate for those parts of a specification where the cost of undetected inconsistencies is high.

4 An example

In this chapter we present an extensive example. As a context, we use the specification of a department library system with self-service terminals for borrowing and returning books. The goal and some constraints for the system are given below. In this paper, we model the requirements for borrowing books only.

Sample Application: The Department Library System

Goal: The system shall support a department library, where students themselves can take books from the shelves, read them, borrow or return books, and query the library catalog. Self-service terminals shall be used for borrowing and returning books.

Constraints: Users identify themselves with a personal library card. Every book has a barcode label. Additionally,

there is an electronic safety device located under that barcode label which can be turned on and off by the system. At the exit of the library, a security gate sounds an alarm when a user tries to leave the library with a non-checked-out book. A user may not borrow a book if she has currently borrowed books that are overdue. A book may not be borrowed if it is reserved for another user.

Type scenario:	Borrow books	Version: 1
Actor:	User	
Precondition:	User has personal library card	
Started by:	User wants to borrow books	

Normal flow:

1 User scans her library card
  System validates the card, returns the card, displays user data, displays 'Select function' dialog
2 User selects 'Borrow' function
  System displays 'Borrow' dialog
3 User scans label of book to be borrowed
  System identifies book, creates an 'on-loan' record (consisting of book, user and current date), unlocks safety label, displays book data
4 If user presses 'More books' key,
  System displays 'Borrow' dialog. Repeat step 3
5 User presses 'Finish' key
  System prints loan slip, displays 'Finished' message

Alternative flows:

1a Card is invalid: System returns the card, displays 'Invalid' message (scenario terminates)
2a User has overdue books: System displays 'Denied' message (scenario terminates)
3a Book is reserved for another person: System displays 'Reserved' dialog (scenario continues)

Figure 1. The 'Borrow books' scenario

Figure 1 shows the 'Borrow books' scenario written in a usual structured-text style which is UML-conformant[2]. Figures 2 and 3 present an extract from the class model that is significant for borrowing books. Figure 2 is the class diagram. Figure 3 specifies the states in which a terminal can be.

Whether or not the models specified in Figures 1-3 are consistent, can only be guessed. Here are some examples.

- Consider step 1 of the Borrow books scenario. There is no way to determine systematically where in the class model the functionality of "validates the card" is specified and who knows/composes the "user data" that is required in the reply.
- In step 3 of the Borrow books scenario we do not know whether "creates an 'on-loan' record" is contradictory to the class model (where there is no such class) or whether this requirement has been modeled in different ways in the scenario model and in the class model. Anyway, the description of the 'on-loan' record in the scenario overlaps with the class model as it specifies persistent data.
- We cannot figure out where in the scenario the terminal changes from the idle state to the checking-out state.

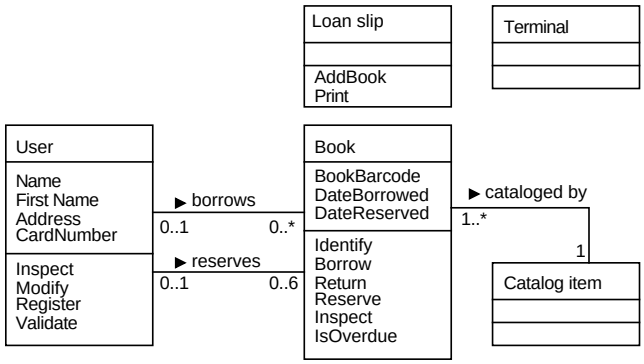

Figure 2. Class diagram representing the classes relevant for borrowing books

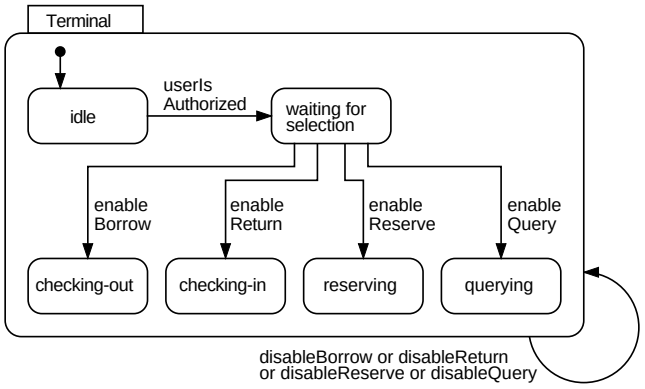

Figure 3. Statechart for Terminal objects

Now we modify the scenario and the class model according to the rules of our lightweight approach to consistency. We remove the overlap by omitting details of the Borrow operation from the scenario and we insert simple references to the class model into the scenario. The class model remains unchanged when employing the simple referencing scheme. The result is given in Figure 4.

When reconsidering the three consistency problems listed above, we find improvements:

- It is now clear that the detailed requirements for card validation are specified in the operation User.Validate. However, we still can only guess that "user data" is provided by this operation.
- The potential contradiction between step 3 of the scenario and the class model has disappeared.
- The points in time where a terminal changes its state are now clearly visible in the scenario.

[2] Principally, UML allows any notation for the description of a use case. However, a notation in the style of Figure 1, following a template that has been proposed by Cockburn [7] is typically used with UML.

Type scenario: Borrow books Version: 2
Actor: User
Precondition: User has personal library card
Started by: User wants to borrow books

Normal flow:
1 User scans her library card
System validates the card (↑User.Validate), returns the card, displays user data, enables selection (↑Terminal.userIsAuthorized), displays 'Select function' dialog
2 User selects 'Borrow' function
System enables borrowing (↑Terminal.enableBorrow), displays 'Borrow' dialog
3 User scans label of book to be borrowed
System identifies book (↑Book.Identify), records the book as borrowed (↑Book.Borrow), unlocks safety label, displays book data
4a User presses 'More books' key
System displays 'Borrow' dialog. Repeat step 3
4b User presses 'Finish' key
System disables borrowing (↑Terminal.disableBorrow), prints loan slip (↑LoanSlip.Print), displays 'Finished' message

Alternative flows
1' If Card is invalid (determined by ↑User.Validate): System returns the card, displays 'Invalid' message (scenario terminates)
2' If User has overdue books (determined by ↑User.Validate): System displays 'Denied' message (scenario terminates)
3' If Book is reserved for another person (determined by ↑Book.Borrow): System displays 'Reserved' dialog (scenario continues)

Figure 4. The revised 'Borrow books' scenario with simple cross-references

Type scenario: Borrow books Version: 3
Actor: User
Precondition: User has personal library card
Started by: User wants to borrow books

Normal flow:
1 User scans her library card {**delivers** numberOfCard}
System validates the card {↑User.Validate (**in** numberOfCard, **out** result, **out** currentUser, **out** userData); return card; **check step** 1'; display userData; ↑Terminal.userIsAuthorized; display 'Select function' dialog}
2 User selects 'Borrow' function
System begins check-out session {**check step** 2'; ↑Terminal.enableBorrow; display 'Borrow' dialog}
3 User scans label of book to be borrowed {**delivers** label}
System checks-out book {↑Book.Identify (**in** label, **out** theBook); ↑theBook.Borrow (**in** currentUser, **out** bookData, **out** status); **check step** 3'; unlock safety label; display bookData}
4a User presses 'More books' key
System iterates {display 'Borrow' dialog; **go to step** 3}
4b User presses 'Finish' key
System terminates check-out session {↑Terminal.disable-Borrow; ↑currentLoanSlip.Print; display 'Finished' message; **terminate**}

Alternative flows:
1' Card is invalid {**if** (result = "invalid card") display 'Invalid' message; **terminate**; **endif**}
2' User has overdue books {**if** (result = "valid user with overdue books") display 'Denied' message; **terminate**; **endif**}
3' Book is reserved for another person {**if** (status = "reserved") display 'Reserved' dialog; **go to step** 4; **endif**}

Figure 5. The rewritten 'Borrow books' scenario with elaborate cross-references

In the next step, we demonstrate the elaborate form of cross-referencing. In this case, we rewrite the scenario as presented in Figure 5. The class diagram is extended by including the scenario and the operation invocation references (Figure 6). Operations in the class model have to be specified more precisely now. Figure 7 shows the operation 'Validate' as an example.

In this model, we can derive the contents of the "user data" required in step 1 of the scenario. Furthermore, this model reveals a gap in the specification: when looking at the class Loan slip in the class model, we find that the scenario Borrow books uses the Print operation of Loan slip, but nothing else. Thus the contents of the printed loan slip is unspecified. We can close this gap by creating a new (empty) loan slip in step 2 of the scenario and using the Add operation of class Loan slip either in step 3 of the scenario or in operation Borrow of class Book. Figure 6 has to be updated accordingly in order to reflect the additional references.

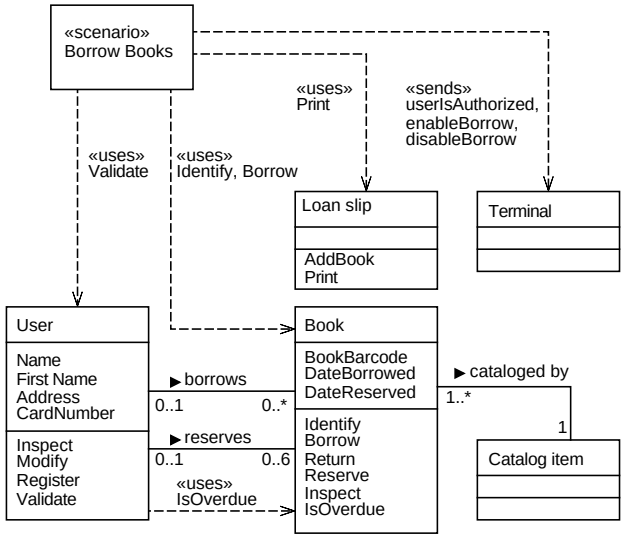

Figure 6. The class diagram extended with stereotypes modeling the 'Borrow books' scenario and cross-references

```
class operation Validate (in numberOfCard: Number, out result:
                  ValidationResult, out id: User, out userData: String)
        in class User
pre     Terminal is in state "idle"
post    if (exists x in User • x.CardNumber = numberOfCard)
          if (for all b in x.borrows • b.IsOverdue = false)
             result = "valid user"; id = x, userData = (x.FirstName,
             x.Name, x.CardNumber)        // x.borrows is the set of all
                                          // books that have been
                                          // borrowed by user x
          else result = "valid user with overdue books"
          endif
        else result = "invalid card"
        endif
end
```

Figure 7. Precise specification of operation Validate of class User

5 Checking consistency

In this chapter, we present a set of rules for checking the consistency between a scenario model and a class model. The rules can be used both constructively for working towards a consistent specification and analytically for checking that a given specification is consistent.

We have two kinds of rules: (a) rules that can be checked automatically by a tool, and (b) rules that guide a person when inspecting a specification.

5.1 Formally checkable rules

Our approach has not been designed with formal verification of consistency in mind. Nevertheless, we have some formally checkable rules. They mainly concern conformance between references and referenced items and completeness of references. When using the simple referencing scheme, only Rule 3 applies. With the elaborate referencing scheme, we additionally have Rules 4 and 5.

RULE 3 (Name consistency). (a) Every operation that is not referenced within the class model (typically in another operation in order to establish the postcondition for this operation) should be referenced in a scenario. (b) Every event in the class model that is triggered in the course of an interaction should be referenced in the scenario where this interaction is specified.

RULE 4 (Conformance of syntax). The syntax of any formal operation invocation or event handling in a scenario must conform to the syntax definition of the operation or event in the class model.

RULE 5 (Conformance of references). Every reference from a scenario to an item in the class model must have a corresponding reference in the extended class model, namely between the scenario stereotype and the class that the referenced item belongs to. The reverse must also be true.

5.2 Rules for inspection

When inspecting a specification that is constructed according to our lightweight approach, we have three rules that guide inspection for inter-model consistency. Rule 1 (see Section 3.2) checks the specification for unnecessary redundancy. Rules 6 and 7 check the absence of contradictions and of partial incompleteness. Rule 6 is applicable with the simple referencing scheme, whereas Rule 7 works only with the elaborate one.

RULE 6 (Correspondence of contents). For every step of a scenario, assess whether the required response is completely specifiable locally within the scenario (that means, the response is neither state-dependent nor requires a state change). When this is not true, make sure that

(a) for every non-local action a reference to the class model is in place,
(b) the items referenced in the class model provide the information, state change or actions that are required to produce the response.

RULE 7 (Flow of information). For every reference from a step in a scenario to the class model, check the flow of information:

(a) Does the reference provide all information from the scenario that is required in the class model to produce a response?
(b) Does the information returned by the referenced items(s) suffice to specify the required response in the scenario?

6 Conclusions

6.1 Achievements

We have presented a lightweight approach to inter-model consistency between a scenario model and a class model that is directly applicable to mainstream object-oriented specification techniques and improves the quality of these specifications. The concepts behind this approach are fairly simple. Nevertheless, we think our approach is a real improvement compared with the current state of practice.

- We can systematically identify information in a class model that corresponds to information in a scenario and vice-versa.
- We have elementary conformance rules that can be checked automatically.
- We have rules for inspecting corresponding information and can systematically detect both contradictions and information that is missing on either side (partial incompleteness).

6.2 Validation

Our approach has not yet been tested with real-size industrial specifications. However, we have done a small validation experiment [12] with graduate and PhD stu-

dents. The goal of this experiment was to compare models with and without cross-references with respect to their acceptance both for writing consistent specifications and for consistency checking.

Based on the example given in this paper, we set up three specifications of the same requirements. Specification A consisted of Figures 1, 2 and 3, representing a conventional UML model collection. Specification B employed the simple scheme of cross-referencing and consisted of Figures 4, 2 and 3. Finally, specification C employed the elaborate scheme and consisted of Figures 5, 6, 3 and 7.

Twelve students participated in the experiment. They first received the three specifications and had to answer some questions about each of them (to make sure that they thoroughly studied and understood the models). Then they had to answer questions about their preferences concerning the three types of specifications. Figures 8 and 9 present some results. Due to the small number of participants, we do not calculate statistics. However, the figures show a very clear preference for specifications using either the simple or the elaborate cross-referencing scheme.

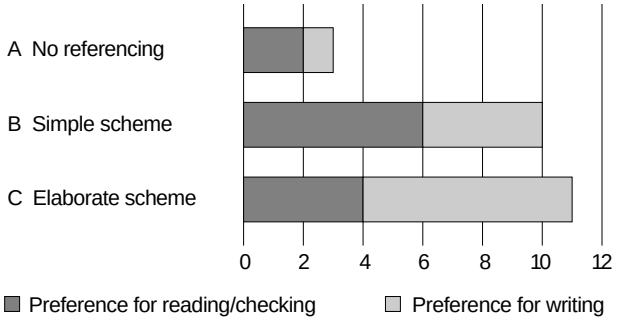

Figure 8. Validation results: overall preferences of participants

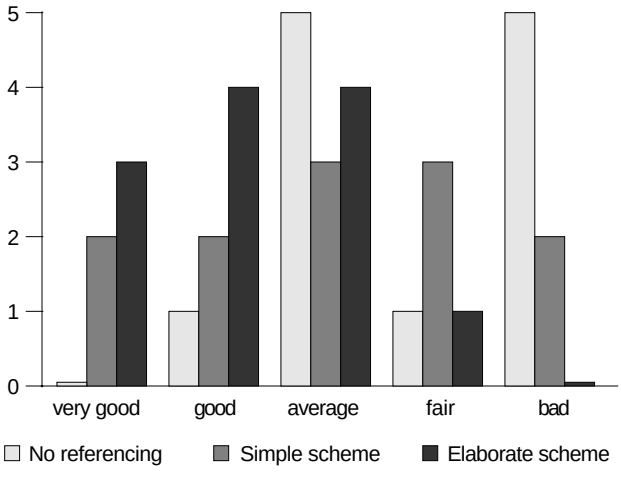

Figure 9. Validation results: suitability for checking consistency

6.3 Towards a stronger variant of consistency

With stronger assumptions about the underlying scenario model and class model, a stronger approach to inter-model consistency is also possible. Within the framework of our ADORA project, we have developed some preliminary ideas for such a concept [11], [13]. The basic idea is to have an integrated model of interaction, structure, functions and behavior. Such a model can be developed by taking the concept behind the elaborate referencing scheme (illustrated in Figure 6) a step further. Instead of "stereotyped" references we generalize the notion of objects. We envisage an object model (not a class model) having three kinds of objects.

- *Plain objects* model both public and hidden data and the operations on this data.
- *State-enhanced objects* are plain objects that additionally have an explicit specification of the states and state transitions that determine the behavior of the object during its lifetime.
- *Scenario objects* model user-system interaction with scenarios that are semi-formally expressed with a statechart-based language.

We view this integrated model *extensionally*: An object in the object model is a representative of a single concrete instance. In order to model a collection of instances, we use an *object set*. (In the example of Chapter 4, the scenario Borrow Books would be a scenario object. Book and Terminal would be object sets, the latter being a state-enhanced one.)

The reference links of the elaborate referencing scheme are replaced by *messages* that model operation invocation, event occurrence, attribute referencing, etc. Thus we can have all aspects of a functional specification within one single model. Consequently, the syntactic consistency problems found in a model collection disappear. The semantic consistency problems become easier to handle; consistency rules can partially be built into the metamodel. Cross-referencing is no longer necessary.

In order to keep the diagrams that represent the model to a reasonable size, a hierarchical decomposition of the model is required. Due to a stringent consistency concept for the underlying model, partial views of the model can be generated which are consistent with each other by construction.

6.4 Open problems and future work

Our approach does not deal with inconsistency management; that means, with the problem of deliberately living with inconsistency in a specification. However, we think that a concept for dealing with inter-model consistency is a prerequisite for any approach to managing inconsistency in a controlled way.

For the future, we plan to pursue our lightweight approach by using it in practical applications and to assess its usefulness when applied by practitioners.

At the same time, we will continue the development of an integrated model for specifying functional requirements in the framework of our ADORA project (cf. last paragraph of Section 2.4). As far as consistency is concerned, we plan to explore the middle ground between the lightweight approach presented in this paper and the formal approaches of other researchers mentioned in the introduction.

References

[1] Berner, S., Joos, S., Glinz, M. Arnold, M. (1998). A Visualization Concept for Hierarchical Object Models. *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE-98)*. 225-228.

[2] Berner, S., Glinz, M., Joos, S. (1999). A Classification of Stereotypes for Object-Oriented Modeling Languages. *Proceedings of the Second International Conference on the Unified Modeling Language*, Fort Collins. Berlin, etc. Springer.

[3] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*, 2nd ed. Redwood City, Ca.: Benjamin/ Cummings.

[4] Buhr, R.J.A. (1998). Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering* **24**, 12 (Dec. 1998). 1131-1155.

[5] Carroll, J.M. (ed.)(1995). *Scenario-Based Design.* New York: John Wiley&Sons.

[6] Coad, P., Yourdon E. (1991). *Object-Oriented Analysis*. Englewood Cliffs, N. J.: Prentice Hall.

[7] Cockburn, A. (1997). Using Goal-Based Use Cases. *Journal of Object-Oriented Programming* **10**, 7. 56-62.

[8] Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. (1994). Inconsistency Handling in Multi-Perspective Specifications. *IEEE Transactions on Software Engineering* **20**, 8 (Aug. 1994). 569-578.

[9] Firesmith, D.C. (1994). Modeling the Dynamic Behavior of Systems, Mechanisms and Classes with Scenarios," *Report on Object Analysis and Design*, **1**, 2. 32-36&47.

[10] Firesmith, D., Henderson-Sellers, B. H., Graham, I., Page-Jones, M. (1998). *Open Modeling Language (OML) – Reference Manual*. SIGS reference library series. Cambridge, etc.: Cambridge University Press.

[11] Glinz, M. (1995). An Integrated Formal Model of Scenarios Based on Statecharts. In Schäfer, W. and Botella, P. (eds.): *Software Engineering – ESEC '95. Proceedings of the 5th European Software Engineering Conference*, Sitges, Spain. Lecture Notes in Computer Science 989, Berlin, etc.: Springer. 254-271.

[12] Glinz, M., Schett, N. (2000). *Preliminary Validation of a Lightweight Approach to Consistency of Scenarios and Class Models*. Research report, Institut für Informatik, University of Zurich. http://www.ifi.unizh.ch/groups/req/ftp/papers/crossref_validation.pdf

[13] Glinz, M., Berner, S., Joos, S., Ryser, J., Schett, N., Schmid, R., Xia, Y. (2000). The ADORA Approach to Object-Oriented Modeling of Software. *Submitted for publication*. http://www.ifi.unizh.ch/groups/req/ftp/papers/ADORA_approach.pdf

[14] Grundy, J., Hosking, J., Mugridge, W.B. (1998). Inconsistency Management for Multiple-View Software Development Environments. *IEEE Transactions on Software Engineering* **24**, 11 (Nov. 1998). 960-981.

[15] Heitmeyer, C., Kirby, J., Labaw, B., Archer, M. Bharadwaj, R. (1998). Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications. *IEEE Transactions on Software Engineering* **24**, 11 (Nov. 1998). 927-948.

[16] Hsia, P., Samuel, J., Gao, J., Kung, D. (1994). Formal Approach to Scenario Analysis. *IEEE Software* **11**, 2 (March 1994). 33-41.

[17] Hunter, A., Nuseibeh, B. (1998). Managing Inconsistent Specifications: Reasoning, Analysis and Action. *ACM Transactions on Software Engineering and Methodology* **7**, 4 (Oct. 1998). 335-367.

[18] Jackson, M.A. (1995). *Software Requirements and Specifications : A Lexicon of Practice, Principles and Prejudices.* Wokingham, etc.: Addison-Wesley.

[19] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G. (1992). *Object-Oriented Software Engineering – A Use Case Driven Approach*. Reading, Mass., etc.: Addison-Wesley.

[20] Joos, S., Berner, S., Arnold, M., Glinz, M. (1997). Hierarchische Zerlegung in objektorientierten Spezifikationsmodellen [Hierarchical Decomposition in Object-Oriented Specification Models (in German)]. *Softwaretechnik-Trends*, **17**, 1 (Feb. 1997), 29-37.

[21] Kösters, G., Pagel, B.-U., Winter, M. (1997). Coupling Use Cases and Class Models. *BCS FACS/EROS Workshop on Making Object-oriented Methods more Rigorous*. London http://www.informatik.fernuni-hagen.de/import/pi3/publikationen/abstracts/ROOM.ps.zip

[22] Kruchten, P. (1998). *The Rational Unified. Process: An Introduction*. Reading, Mass., etc.: Addison-Wesley.

[23] van Lamsweerde, A., Darimont, R., Letier, E. (1998). Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering* **24**, 11 (Nov. 1998). 908-926.

[24] Martin, J. (1989-90). *Information Engineering: A Trilogy.* Englewood Cliffs, N.J.: Prentice Hall.

[25] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. (1991). *Object-Oriented Modeling and Design.* Englewood Cliffs, N. J.: Prentice Hall.

[26] Rumbaugh, J., Jacobson, I., Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Reading, Mass., etc.: Addison-Wesley.

[27] Yourdon, E. (1989). *Modern Structured Analysis.* Englewood Cliffs, N.J.: Prentice Hall.