Published in Proceedings of the 4th India Software Engineering Conference (ISEC '11). ACM, New York, NY, USA, 135-144.
DOI=10.1145/1953355.1953375 http://doi.acm.org/10.1145/1953355.1953375

Use-Case and Scenario Metamodeling for Automated Processing in a Reverse Engineering Tool

Julien REPOND, Philippe DUGERDIL
Dept. of Information Systems
HEG - Univ. of Applied Sciences
7 route de Drize, CH-1227 Geneva, Switzerland
+41 22 388 17 00
julien.repond@hesge.ch
philippe.dugerdil@hesge.ch

Pietro DESCOMBES
HORTIS GRC SA
12 avenue des Morgines CH-1213 Geneva, Switzerland
+41 22 860 84 60
pietro.descombes@hortis.ch

ABSTRACT

The reverse engineering methodology we developed is based on the reverse specification of the use-cases linked to the execution trace of the legacy system. Basically we aim at recovering the traceability links between the robustness model that represents the analysis of the use-case and its actual implementation classes. Therefore we need to be able to edit the use-cases and the scenarios of the system so that the environment could process this information together with the robustness model and the execution trace to recover the traceability links. We then developed a use-case and scenario editor that is coupled to a robustness model editor. In this paper, we present the UML meta-model extensions we made to formalize the use case and scenario models. Then we present the techniques we developed to assure the coherence between both models. Next we present the way we link the use-case and scenarios to the robustness model and present the Eclipse-based tool we developed. The key contributions of the paper are the definition of the use-cases and scenarios meta-models, the link between the specification and analysis meta models and the mechanisms we developed to assure their mutual coherence. Finally, we present the way these models can be edited and processed in the context of a real tool.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications – *languages, methodologies, tools.*

General Terms

Design, Experimentation, Algorithm, Standardization.

Keywords

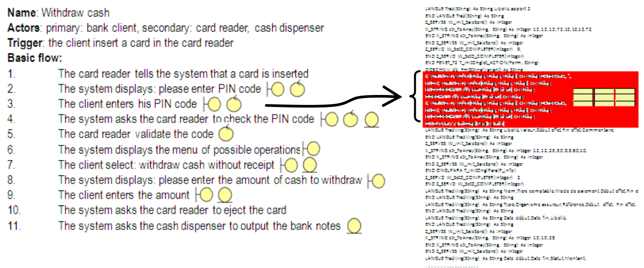
Use case formalization, software specification, use-case modeling, reverse engineering.

1. INTRODUCTION

Our work on reverse-engineering [7][8] of legacy systems is based on the Unified Process (UP) [12] which is a use-case driven software development process whose models are designed using the UML. The iterative and incremental reverse-engineering technique we developed starts from the recovery of the use-cases of the system. Then with the help of the analysis models, we can later incrementally re-create the traceability links between the functional specification and the source code of the system. In summary, our reverse-engineering technique works through the following steps:

1. Re-document the system use-cases;
2. Design the Unified Process' robustness (analysis) diagrams associated to each of the use-cases [14];
3. Execute the system according to the use-cases and record the execution trace;
4. Analyze the execution trace and identify the classes involved in the trace;
5. Map the classes in the trace to the objects of the robustness diagram;
6. Re-document the architecture of the system by clustering the classes based on their role in the implementation of the use-case.

The key technique in our methodology is to link the steps of the scenarios to the segments of the corresponding execution trace. This lets us know what implementation classes are involved at what step. Since, by use-case analysis [14], we know what robustness object is involved at what step of the use-case, we can eventually link the implementation classes to the corresponding robustness objects. Figure 1 illustrates the central idea of the method. On the left, a use-case flow is presented with the robustness objects (boundary, entity and control objects [13][14][1][2]) associated to each step.



Name: Withdraw cash
Actors: primary: bank client, secondary: card reader, cash dispenser
Trigger: the client insert a card in the card reader
Basic flow:
1. The card reader tells the system that a card is inserted
2. The system displays: please enter PIN code
3. The client enters his PIN code
4. The system asks the card reader to check the PIN code
5. The card reader validate the code
6. The system displays the menu of possible operations
7. The client select: withdraw cash without receipt
8. The system displays: please enter the amount of cash to withdraw
9. The client enters the amount
10. The system asks the card reader to eject the card
11. The system asks the cash dispenser to output the bank notes

Figure 1. Reverse engineering principle

These represent the analysis objects involved in the realization (in UP parlance) of the use-case. On the left we present the corresponding execution trace as a list of method calls. Since each method belongs to an implementation class we can link the robustness objects to these classes by identifying the segment of the trace that corresponds to the step of the use-case [7][8]. However, Figure 1 is actually a short cut of the method. In fact, we do not compare the use-case itself to the execution trace, but an instance of the use-case i.e. an actual scenario performed by a user of the system. Therefore, what we try to match with the execution trace are the steps of the scenario that generated the trace. Hence, in the reverse engineering environment we are building, we need to include both a use-case and scenario editor. Therefore we must have a clear model of how use-cases and scenarios depend on each other. In the UML, the Use-Case Model only represents the overall view of all the use-cases of a system with their actors and relationships. However, the flows are not formally defined. Here is what we can read in the UML specification: *"The behavior of a use case can be described by a specification that is some kind of Behavior [...] such as interactions, activities, and state machines, or by pre-conditions and post-conditions as well as natural language text where appropriate. [...] Which of these techniques to use depends on the intended reader [...]"* [17]. In this statement the "behavior" of the use-case is to be understood as the description of the interactions between the system and the actors. In this paper, we present a formalization of the use-case flows and use-case relationships as well as the formalization of use-cases instances (scenarios) as extensions of the UML meta-model. Moreover, since we must match the robustness objects to implementation classes based on their involvement in the step of the use-case, we must be able to attach some analysis objects to the steps of the use-cases. The corresponding meta-model will then be described next. Finally we present the tool we developed to edit the use-case and scenarios of a legacy systems to reverse-engineer and the way we can link them to the robustness diagram objects.

The paper is structured as follows. Section 2 presents the extension of the UML use-case meta-model. Section 3 presents the way the scenarios are modeled and associated to the use-cases. In section 4, the link between the use-cases, scenarios and the objects of the robustness model is explained and formalized. A case study is presented in section 5 together with the tool we developed. Section 6 presents the related work and section 7 concludes the paper.

2. META-MODEL

2.1 Informal use-case structure

First of all, it is worth mentioning that the UML specification doesn't define any informal, even less formal, structure for use case writing. Consequently different developers may use different ways to represent the "behavior" of the use-case. When trying to formalize the use-cases, the first step is therefore to choose the informal representation to start from. Fortunately best practices have emerged [4][5] that are of common use today. Then we based our formalization on these recommendations and tried to keep it as simple as possible. Following these common practices, the use-case attributes we kept are presented in Table 1. In the latter, the expression "use-case execution" is a short cut to mean: the execution of a scenario that conforms to the description of the use-case to which it belongs. Although the trigger, pre and post conditions are generally expressed by a natural language sentence, or set of sentences, there are several options for the flows. Indeed not all the authors have the same understanding of the way it should be specified.

Table 1. Use-case attributes

Attribute name	Semantics
Name	Name of the use-case
Trigger	Action that will trigger the execution of the use-case
Pre-condition	System state expected before the use case execution begins
Post-condition	System state reached after the use case has been executed
Primary actor	Name of the actor who will benefit from the execution of the use-case
Secondary actors	Names of the actors who are involved in the execution of the use-case but will not get the benefit from it
Main flow	Sequence of interaction between the system and the actors representing the most common use-case execution
Alternative flow	Additional interactions between the system and the actors representing variants from, or errors in, the main flow

For some, it should be expressed in a formal way like UML's Activity Diagrams [15], Statecharts [21], Petri Nets or even formal languages. However, we do believe that use-case specifications should keep their natural language form, because this is the best way to communicate with the customers. However, even with the textual form there are variants. For example, some people advocate the use of full paragraphs of text while some others prefer to specify a sequential collection of numbered free text sentences [5] or structured sentences with limited vocabulary [20]. Each of these sentences represents one step in the flow [5]. Today, the consensus is growing around the idea of a collection of numbered sentences. Besides, the location of the alternative flows is also variable among authors. Some of them place them directly within the main flow with a conditional statement. Others gather them after the end of the main flow as a separate section of the use-case. The latter is the most convenient structure since this makes the reading of the main flow easier. Following Cockburn [5][6] we chose to represent the flows as separate numbered free-text sentences (the steps) and to group the alternatives at the end of the use-case. Since the alternatives are separated from the main flow, there is some extra information to attach to each alternative. First, the execution condition must be specified i.e. in what situation would the alternative be triggered. This is generally represented in natural language. Second we must identify the step of the main flow that each alternative extends and the step of the main flow at which the processing will resume. Third we must indicate whether the alternative flow replaces the main flow step or complements it. Finally, there is an issue with the ending of the scenarios since the end of an alternative flow may mean the end of the scenario. This must also be indicated. In summary, our use-case flows are structured as two sections. First we represent the main flow as a list of numbered steps. Second, we represent all the alternative flows as lists of numbered steps together with their extra information. For an alternative, if the step at which to resume the processing is not indicated, the scenario will end at the end of the alternative. Table 2 presents the attributes of the alternative flows we kept in our model.

Table 2. Alternative flow attributes

Attribute name	Semantics
Name	Alternative flow name together with the trigger condition.
Extended step	Main flow step that is extended / replaced by the alternative flow. In case the extended step is not indicated, this means that the alternative could happen anywhere within the main flow (like a "cancel" event).
Replace	Boolean indicating if the alternative flow replaces the main flow step or extends it
Back step	Identification of the main flow step at which the processing must resume after the execution of the alternative flow. In case the back step is not indicated, the scenario will end at the end of the alternative flow.
Flow	Sequence of interaction between the system and the actors.

2.2 Formalization of the use-case structure

In this subsection we present the formalization of the use case flows as extensions of the UML meta model. As shown in Figure 2, a flow is associated to a collection of *Steps*. The *Flow* class is abstract because each flow must be clearly typed as main or alternative. A step represents an elementary interaction or event. It contains a description (i.e. a sentence describing the correspond-ding elementary action) and a number for the user to be able to identify it visually (not showed on the diagram). Following Cockburn's advice [5], alternative flows do not have alternatives themselves. In all the models below, the darker (red) boxes are original classes of the UML meta model.

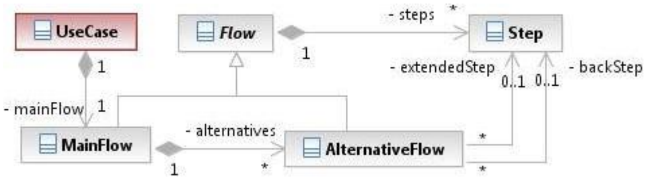


Figure 2. Use case flows meta model

Alternative flows are particular cases differing from the default behavior. The *extendedStep* and *backStep* of the alternative flow have been represented as directed association to the *Step* class. The cardinality of these associations are 0..1 to allow the modeling of alternative flows that could be triggered anytime. For example, this is the case of the "Cancel" event triggered by the user of the system. Therefore, if the *extendedStep* is not indicated, this means that the alternative can occur at any time. On the other hand if the *backStep* reference is empty, this means that the current scenario ends at the end of the alternative flow. Figure 3, represents the *Step* class hierarchy. There are 4 types of steps categorized in 2 sub hierarchies. First, *AlterableStep* represents the events that can be executed and possibly extended by alternative flows. Second, *NonExecutableStep* represents entries in the use-case flow that cannot be executed. This includes labels and control information. *ActionStep* is the most common event which represents an action executed by an actor or the by system to meet the goal of the use case [5]. *InclusionStep* represents the location where the flow of an included use-case can be inserted (see section 2.3 below).

GeneralizationStep is used for the modeling of use-case generalization hierarchies. It defines the location at which the flows of a specialization use case must be inserted in the flow of a generalized use-case (see use-case relationship below).

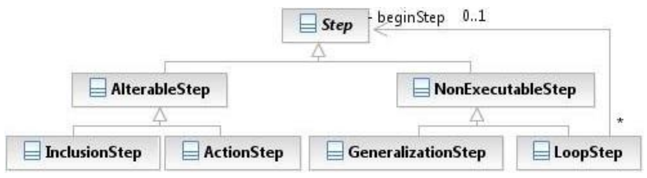


Figure 3. Flow step hierarchy

Finally, *LoopStep* represents control information to specify step repetitions. This information takes place after the block of steps that must be repeated. Its reference to *beginStep* identifies the first step of the block that must be repeated.

2.3 Use Case Relationships

Beyond the specification of isolated use-cases, UML allows to represent use-case relationships to help with the reuse of specifications and to model asynchronous extension of use-cases. However this facility should be used with care since, according to Cockburn [5], a common mistake made by specification engineers is to invest too much time and energy to define use-cases relationship rather than focusing on the textual content (the flows). In the worst case, the end result would be a complex structure of use-cases linked to each other whose global meaning would be obscure to the customer. Nonetheless the wise use of use-case relationship does provide a powerful way to reduce their complexity by isolating parts of behavior which are performed only in certain circumstances [4]. Therefore we decided to support the use-case relationship in our meta-model. The key contribution here is to model exactly where in the target flow the included, extended or specialized use-case flows must be inserted, while limiting as much as possible the impact on the original UML metamodel. Several authors [3][11][15][20][23] acknowledged the need to model relationships explicitly. However none of them, but Zelinka at al. [23] referenced explicitly the step at which an insertion flow must be inserted. However, the impact on the UML metamodel of Zelinka's et al. proposal is much bigger than ours. In fact they introduced an extra relation called *FlowInclusion* that links together a flow (set of steps), an inclusion step and an include relationship. However we consider this modeling awkward since the UML *Include* relationship does represent the inclusion flow (since it relates two use-cases that themselves have flows). What is needed is simply a reference from the UML's *Include* relationship to the inclusion step, as presented in the sub section below.

Inclusion relationship

Fundamentally, the inclusion relationship allows the specification engineer to extract a subsequence of events that is common to several use-cases and to create a use-case of its own: the inclusion use-case (sometimes called a sub function level use case [5]). Then, the writing of the original use-cases will be simplified since the repeated subsequence will be replaced by an include statement [14]. The metamodel representation of the *Include* relationship is presented in Figure 4. What we added to the UML meta model is the association with the *InclusionStep*, since the concept of step is absent from the UML meta model.

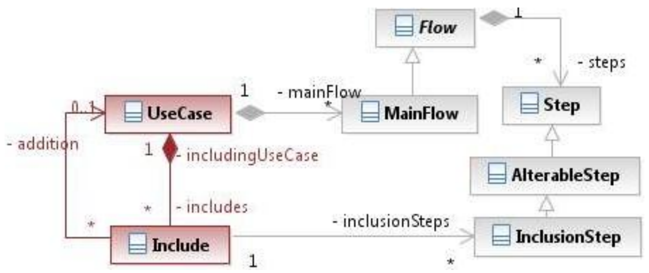


Figure 4. Addition to the UML meta model for inclusions

InclusionStep represents the location in a flow where the set of steps of an inclusion use-case will be inserted (Figure 3). Such a step is executable since the steps of the included use-case will indeed be executed. The included use-case could be inserted as well in the main flow as in an alternative flow. The UML *Include* relationship lets us know which use-case is included. Since a use-case flow can contain more than one inclusion step referencing the same included use-case, the *Include* relationship can be associated with several *InclusionSteps*. But each one of the latter must reference one and only one *Include* instance. The name of an inclusion step is composed of the keyword *include* followed by the name of the use-case to include [4][5]. For example, if a use case *Pay an invoice* requires the user to be authenticated, we could have the following partial flow, where *Authenticate user* represents an included use-case:

```
Pay an invoice
include "Authenticate user"
Customer selects an invoice.
Customer fills the amount in.
System validates the amount.
…
```

Extension relationship

The extension relationship (*Extend*) between use cases is an extra UML relationship that is much less used than the "include" relationship. Since its semantics has long been unclear, experts used to recommend not using it [5][18]. Since UML 2.0 however, this semantics has been better defined. In fact an "extension use-case" represents some extra behavior added asynchronously to a target use case [17]. But the extension use-case is optional and the extended use-case can be delivered without the extension use-cases. In contrast, the included use cases are not optional and must be delivered with the included use-case. The insertion is controlled by a trigger condition. During use-case execution, that condition is evaluated and if satisfied, the flow of the extension use-case is inserted in the target use-case at a specific location called the extension point. The latter is modeled by the *ExtensionPoint* class. The extension point owns a reference to the flow step after which the flow is inserted, called *referenceStep*. Once the execution of the extension use-case is completed, the execution is resumed just after the extension point (i.e. after the step referenced by the extension point) in the extended use-case flow [5][14]. Since the extended use-case can be delivered without the extensions use-cases, the definition of an extension to the use-case should be as little intrusive as possible. Figure 5 presents the meta model of the *Extend* relationship.

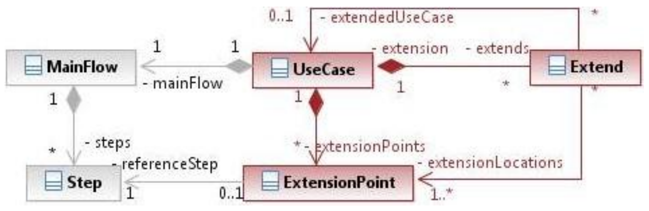


Figure 5. Addition to the UML meta model for extensions

What we added to the UML meta model is the association from the *ExtensionPoint* to the *Step* since, again, the concept of step is absent from the UML meta model. It must be highlighted however that our interpretation of the extension relationship differs from the informal UML specification [17], in order to get closer to the original idea of Jacobson [13][14]. In the UML specification, an extension use-case is described as an incomplete use-case made of a collection of use-case fragments. Each fragment is a piece of behavior (set of steps). The number of fragments in an extension use-case must comply with the number of insertion points in the target use-case. When triggered, the extension use-case will insert each of its fragments to the corresponding insertion point of the target use-case. From this explanation, it is clear that extension use-cases are very different from the other kind of use-cases. Therefore, we preferred to keep a unified definition of the use-cases for the sake of simplification. Then, an extension use-case has the same structure as any other use-case with a main flow and possibly alternative flows. Such a flow will then be inserted at some specific insertion point in the target use-case identified by the *Extend* relationship. The multiplicity of the *ExtensionPoint* end of the *Extend-ExtensionPoint* association accounts for the multiple locations at which a given extension use-case could be inserted in the target use-case.

Generalization relationship

The generalization relationship is even less common than the extension relationship. It is intended to model families of similar use-case whose flows only differ by a few steps [4][5]. This is the equivalent of the "template method" design pattern in class diagrams: some incomplete global behavior is specified in an abstract class and the specific parts to complete it will be defined in its specializations. Similarly the flow of a parent use-case (generalization) will have most of its steps defined but a few specialized steps to be specified in its child use-cases (specializations). In contrast with the inclusion relationship, the generalization relationship is used when most of the behavior is common among a set of use-cases. Jacobson called the child use case a *concrete use case* because it is complete and can be executed. In fact it will inherit most of its behavior (steps) from the parent use-case while adding a few more steps. On the contrary, Jacobson called the parent use case an *abstract* or *semi-*

manufactured use case because its flows are incomplete. Therefore the latter cannot be executed alone. Jacobson even says that the parent use-cases only exist to be reused [14]. Figure 6 presents a conceptual view of the execution of a specialized use-case, inspired from [11]. In this example, the execution starts in the flow of the child use-case, then goes to the parent flow (flow inheritance) then back to the child's and finally ends with the parent flow.

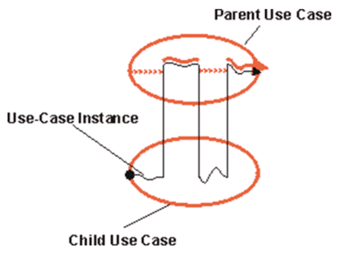


Figure 6. Execution flow in specialized use case

Since the generalization relationship between use-cases is not part of the UML specification, we had to define the whole use-case generalization meta-model based on the informal advice of use-case experts [14][5][4]. First, a generalized use-case must define the locations in its flows (that we called *GeneralizationPoint*) where the specialized subflows must be inserted. Second, the specialized use-case must define the subflows (that we called the *SpecializationFlow*) to insert at each of these locations. The child use case cannot exist without its parent use-case, and **must** define one subflow for each of the generalization points. In the meta-model, the *UseCase* class is specialized in two subclasses, the *MainUseCase* that represents standard non-specialized use-case and *SpecializationUseCase*. A *MainUseCase* contains a single main flow. In contrast, a *SpecializationUseCase* contains a collection of *SpecializationFlow*, each of them representing a single subflow to insert at some *GeneralizationPoint* in the flow inherited from its parent.

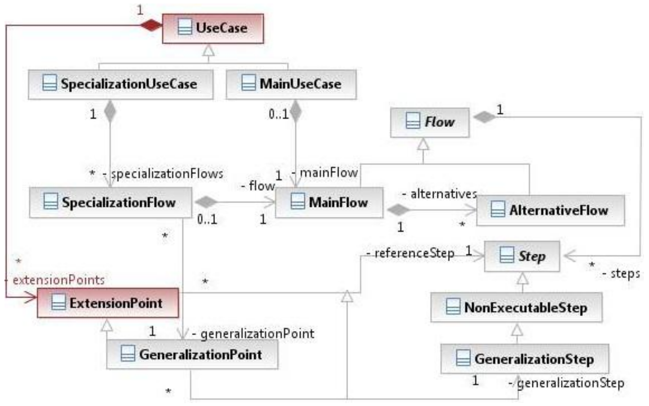


Figure 7. Generalization meta model

This is why each *SpecializationFlow* is associated to a single *GeneralizationPoint.* Since the insertion mechanism is similar to the one of extension use-case, the *GeneralizationPoint* is a specialization of the *ExtensionPoint.* However, the *referenceStep* that represents the step at which an extension is inserted in some target use-case is replaced by the *GeneralizationStep* that is a non-executable step (i.e. a label in the flow of the generalized use-case, see Figure 3). Finally, each of the *SpecializationFlow* has the same structure as the main flow of a main use case. Especially it could have alternatives. This is presented in Figure 7.

3. USE CASE INSTANCE (SCENARIO)

During the last decade, the difference between a use case and a scenario was not clearly defined. Quoting Cockburn: *"[...] it seemed no one could say what a use case was, or name the difference between a use case and a scenario, the basic, attractive idea remained: write a short, textual description of how a system interacts with its surroundings while performing a function of value to one of its users [...]"* [6]. However, long ago Jacobson already proposed a useful analogy. The use-case is the equivalent of a class of behavior and the execution of a use-case represents an instance of the use-case [13]. In our reverse engineering context this distinction must be formalized and we adhere to the idea of Jacobson. Being an instance of a use-case, the scenario represents a specific path among all the possible flows of the use-case. It is a definite sequence of steps with specific values for each input and output information. In Figure 8, we present the steps of a use-case as an directed graph. We call it the use-case graph. Each node represents a step and the arcs represent the possible transitions from a step to the next. The numbers in the nodes are the ones of the steps. A node whose number is composed of 2 sub numbers separated by a dot is part of an alternative flow. The main flow is presented in the central part of the graph (steps: 1; 2; 3; 4; 5) and the alternative flows are presented around it (2.1; 2.2 & 4.1). The multiple paths in this graph represent all the possible executions of the use-case (all possible scenarios). For example, the dotted line represents one possible scenario. It must be noted that the steps represented in such a graph could come from a single use-case or from multiple use-cases such as a parent use-case, an inclusion use-case or an extension use-case. For example, in Figure 8, the steps 1, 2, 4 and 5 could come from a parent use case where step 3 could be defined in a child use-case.

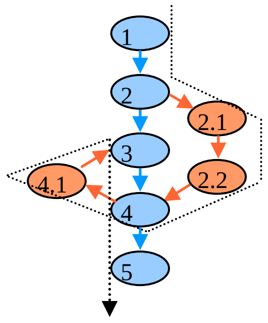


Figure 8. Execution path through use case graph

The meta-model of a scenario is showed in Figure 9. The *Scenario* class represents an instance of a use-case i.e. a path through the use-case graph. This object owns a reference to the use-case it belongs to through the *Scenario-UseCase* association. A *Scenario* is also associated to a collection of *ExecutionLines* that represents the steps of the scenario. The collection is flat since a scenario represents one path through the use-case graph. For example, if the use-case flow contained a loop, the corresponding scenario would contain the actual repetition of steps that would correspond to the exact number of times the loop was repeated. Each scenario step (*ExecutionLine*) owns an association to the step of the use-case it corresponds to. However these two objects are not equivalent since a scenario step includes actual values inputted by the user or outputted to the screen. Besides, we need to be able to freeze executed scenarios at some moment in time to represent executed test cases linked to execution traces. In this situation we must decouple the scenario from the use-cases. This is yet another reason why *ExecutionLine* is separated from *Step* in the meta model and also why the cardinality of the *step* side of the association is 0..1.

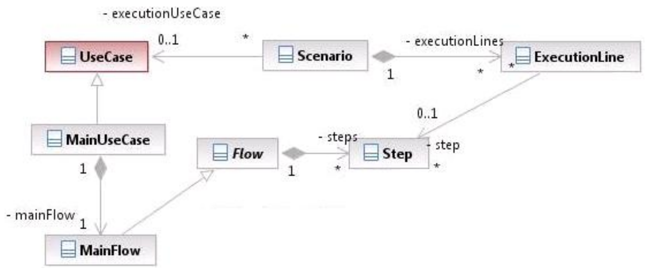
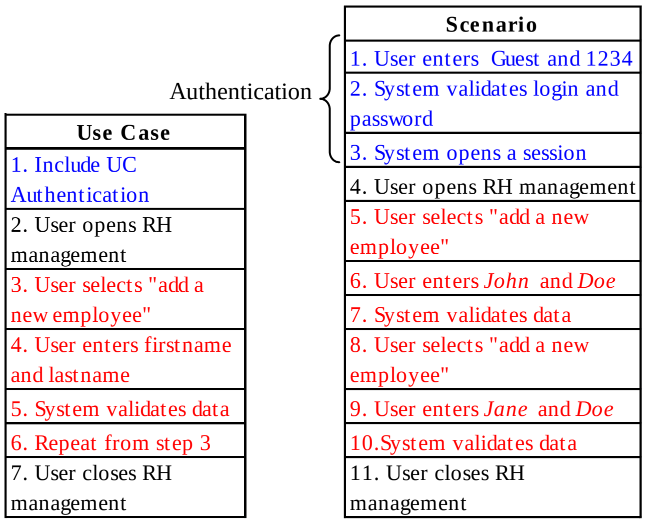


Figure 9. Scenario meta model

The difference between use-case and scenario steps is illustrated in Figure 10. The first step of the use-case represents the inclusion of another use-case (*Authentication*). Therefore, the steps of the latter are located at the beginning of the scenario. Likewise, the loop in the use-case identified by the non-executable step: *6.repeat from step 3* has lead to the repetition of the steps 3-5 twice in the scenario.



Use Case
1. Include UC Authentication
2. User opens RH management
3. User selects "add a new employee"
4. User enters firstname and lastname
5. System validates data
6. Repeat from step 3
7. User closes RH management

Authentication

Scenario
1. User enters Guest and 1234
2. System validates login and password
3. System opens a session
4. User opens RH management
5. User selects "add a new employee"
6. User enters *John* and *Doe*
7. System validates data
8. User selects "add a new employee"
9. User enters *Jane* and *Doe*
10.System validates data
11. User closes RH management

Figure 10. Use case translation to scenario form

3.1 Path synchronization

When a scenario is edited, it stays connected to the corresponding use-case since its steps must conform to those of the use case. If the use-case is later modified, the scenario must be updated accordingly. In some situation an entire sub path must be resynchronized. This is for example the case if a scenario runs through an alternative flow of the use-case that is later deleted. The editor tool should therefore rebuild a path excluding the alternative, while maintaining the other editing decisions made by the user that are not impacted by the change of the use-case.

Figure 11 illustrates the path resynchronization algorithm implanted in our editing tool.

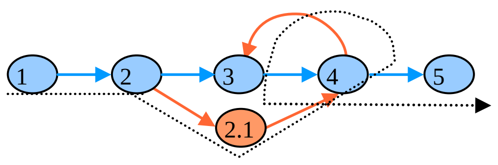


Figure 11. Two alternatives path

The figure presents a use-case graph with an alternative flow (2.1), and a loop through the steps 3 and 4. The dotted line represents a specific scenario which goes through the alternative path and iterates twice on the steps 3 and 4. The sequence of steps in the scenario will be: (1; 2; 2.1; 4; 3; 4; 5). If the user decides to remove the alternative flow (node 2.1) from the use-case, the step must also be removed from all the scenarios. However, since the loop is independent from the alternative path, it must be maintained in the updated scenario. In this case, the recovered path would be: (1; 2; 3; 4; 3; 4; 5). In summary, the path resynchronization algorithm works the following way:

1. Instantiate a new scenario from the main flow only.
2. Compare each node of the new scenario to the old one in sequence.
3. If there is a difference between the nodes, there are two cases:
   a. If the old node comes from an alternative path (alternative flow or loop) that is still available in the use-case. Then the new scenario is updated to go through this alternative path.
   b. If the difference is due to a removed step, the algorithm searches the remaining nodes of the new scenario for the node occurring first in the old scenario. When found the synchronization resumes from this node on. The nodes in the old scenario located between the previous synchronization point and this new synchronization point are discarded.
4. Once the entire path has been rebuilt, the old scenario is replaced by the new one in the editor.

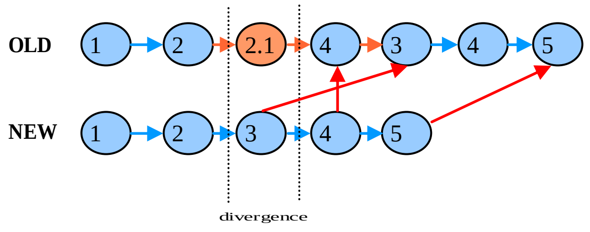


Figure 12. Divergence due to missing node(s)

Figure 12 illustrates the work of the path synchronization algorithm. The new scenario has been instantiated and the first difference is detected at the third node. Since the node 2.1 has been removed, we are in situation 3b of the algorithm. The latter then searches the new scenario for the node occurring first in the old scenario. This search is symbolized by the arrows that link both scenarios. The first node is 4 (it occurs before node 3 in the old scenario). This is the new synchronization point. Next, there is a new difference between the new node 5 and the old node 3

(Figure 13). In this case the algorithm identifies that a loop is the origin of the difference (situation 3a of the algorithm).

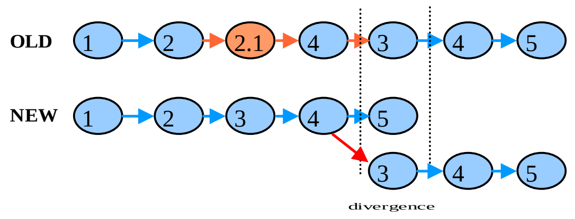
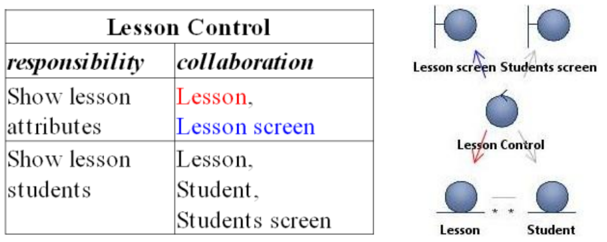


Figure 13. Difference due to a loops

Then, the new scenario goes through the loop as illustrated by the second arrow starting from node 4 in the second graph. From this point, the next nodes are the same between both scenarios and the algorithm ends. In summary, the algorithm has been able to remove a deleted node from the scenario while keeping the loop.

4. LINK TO THE ANALYSIS MODEL

As a quick reminder, the UP [14] as well as some other Agile processes such as Agile Modeling [1] and Iconix [18] advocate the creation of a robustness model as the result of use-case analysis. Such a model contains stereotypes representing the roles the implementation classes will play while realizing the use-case. There are three possible roles (stereotypes): the **entity** object representing the information processed by the use-case, the **boundary** object representing an interface to the outside of the system and the **control** object representing the "coordinator" of the use-case as well as the use-case specific processing [1][2][13][18]. A robustness model is presented on the right of Figure 14. During the design phase of the development process, the robustness model is transformed to an implementation model [14]. Therefore the robustness model bridges the gap between the informal world of the use-case specification and the formal world of the implementation technology [18]. It is built by analyzing each of the steps of the use-case using a responsibility-driven approach with CRC cards [22]. As a result we get the robustness model and a link between each robustness object and the steps in which they are involved. Figure 14 presents the output of such an analysis. On the left we present the CRC card of the control object "Lesson Control". The collaboration displayed in the right column of the table leads to the association between the corresponding objects in the robustness diagram presented on the right of the figure. In forward system engineering, traceability links can be maintained between the implementation classes and the corresponding robustness objects. These links allow identifying quickly the roles of the implementation classes in the system. This greatly helps software understanding. This is exactly why we believe the robustness model to be fundamental in reverse engineering. By recreating the links between the implementation classes and the analysis objects, we could greatly help the maintenance engineer with software understanding. In fact, the latter is known to account for 40-60% of the maintenance effort. Our reverse engineering set of tools will be able to recreate the traceability links between the robustness model objects and the implementation classes, based on the execution trace that correspond to the steps of the scenarios. Therefore, our environment must support use-cases, scenarios and robustness diagrams editing.



Lesson Control	
responsibility	*collaboration*
Show lesson attributes	Lesson, Lesson screen
Show lesson students	Lesson, Student, Students screen

Figure 14. CRC cards and robustness model

Once a use-case is analyzed, the next step is to specify what robustness object is involved at what step of the use-case and when these objects collaborate. This information is then attached to each of the steps of the use-case. Our meta model must therefore allow this information to be explicitly represented. The *Step-Class* association identifies which robustness objects are associated to what *Step* of a use-case flow. The different kinds of robustness objects are represented as specialization of the UML *Class* meta class. This is illustrated in Figure 15.

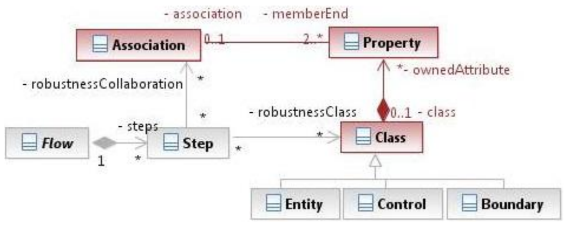


Figure 15. Robustness elements linking

The *Association* class represents a relationship like in a normal UML class diagram. The *Step-Association* association represents the link from a *Step* to the collaborations between the robustness objects that are carried out during this step.

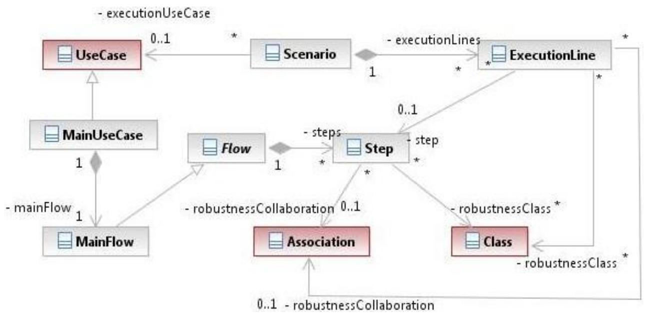
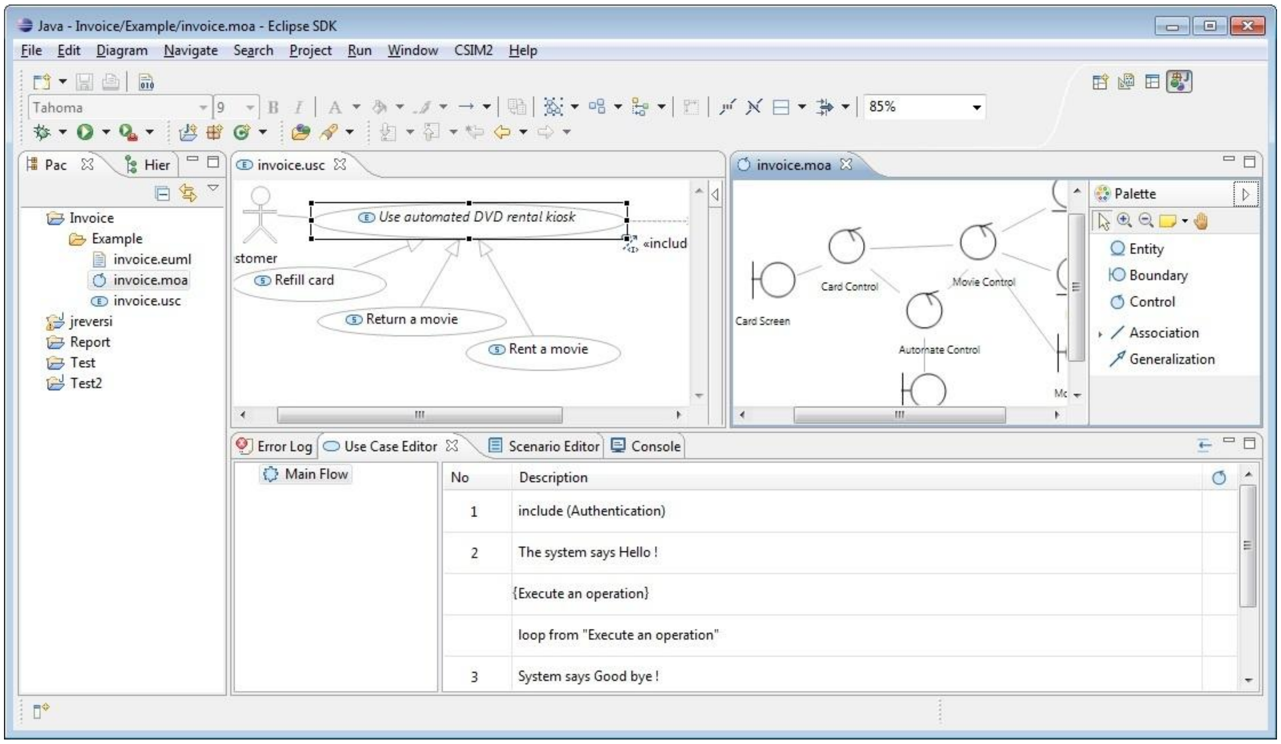


Figure 16. Enhanced scenario meta model

Finally, since the scenarios are derived from the use-cases, their steps must reference the same robustness objects and robustness objects collaboration as the corresponding steps of the use-cases. The extended meta model for scenarios is presented in Figure 16 where the *ExecutionLine* is associated to some *Classes* and *Associations* like the use-case *Step* from which it is derived.

5. CASE STUDY AND TOOL

Our use-case and scenario editor has been developed as an Eclipse plug-in that is based on two projects of the Eclipse foundation: the Eclipse Modeling Framework (EMF) [9] and the Graphical Modeling Framework (GMF) [10]. EMF is a Java based implementation of the Meta-Object Facility (MOF) specification [16]. MOF is a meta-meta-model describing UML. The Eclipse community provides an implementation of the UML notation using EMF. What we did is to extend the corresponding API to integrate our formalization of the use case flows and scenarios. Figure 17, 19 and 20 present some of the screens of our tool.



No	Description
1	include (Authentication)
2	The system says Hello !
	{Execute an operation}
	loop from "Execute an operation"
3	System says Good bye !

Figure 17. Use case editing tool

The user interface proposes two diagram editors (Figure 17), one to edit the robustness diagrams and the other to edit the use-case model. The latter is linked to two extra Eclipse views to display the steps of the use-cases and the steps of the scenarios. They are located at the bottom of Figure 17. The use case model for the case study, a DVD rental kiosk, is presented in Figure 18. There is only one actor, *Customer*, which is the primary actor of all of the use cases. The main use case is *Use automated DVD rental kiosk* which is a generalized use case, specialized by three child use-cases: *Refill card, Return a movie, Rent a movie*. Any operation with the automaton requires the user to be authenticated. This is why the parent use-case includes the *Authentication* use case. Since the parent use case cannot be executed alone it is abstract (name in italics).

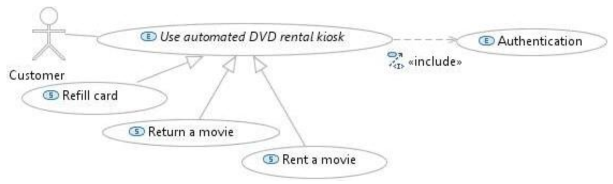
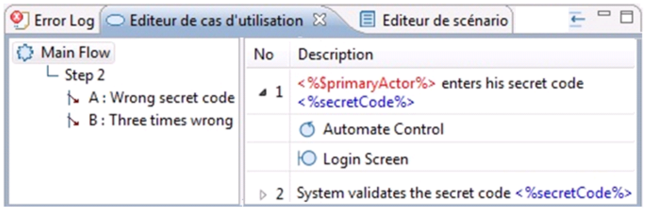


Figure 18. DVD rental kiosk

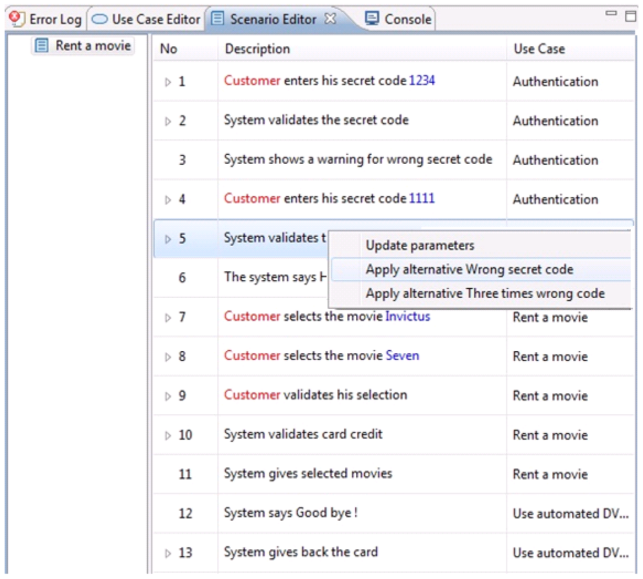
To edit the flow of a use-case we need to select it in the use-case model view. Then its flow is displayed in the editing view as illustrated in Figure 19. This view is split in two columns: the left column shows the flows of the use cases: the main flow, the specialization flow or the alternative flow. The right column shows the sequence of steps corresponding to the flow selected on the left. In the figure we displayed the main flow steps of the use case *Authentication* (see *Main Flow* selected on the left). The editor allows variables to be inserted in the steps so that actual values could be entered in the corresponding scenario (use-case instance) steps. The variables are identified with the <%......%> syntax. Finally the editor lets the user map robustness objects to the step. As we can see, the first step references two robustness objects: *Automate Control* and *Login Screen*. These are the objects which are involved in this step. In the left column we see that step number 2 has two possible alternatives listed below it: first, a wrong secret code is entered, second the user entered a wrong code for three times.



No	Description
1	<%$primaryActor%> enters his secret code <%secretCode%>
	Automate Control
	Login Screen
2	System validates the secret code <%secretCode%>

Figure 19. Use case editing view

When the edition of the use cases is completed, a scenario can be edited as illustrated in Figure 20. In this figure the scenario belongs to the use case *Rent a movie*. Depending on the flows we select using a contextual menu (see Figure 20) the corresponding steps are added to the scenario. Moreover, if some step of the use case involves the inclusion of a use-case, the steps of the latter are automatically added to the scenario. This can be observed in the figure where the steps of the *Authentication* use-case were automatically inserted. Finally the values of the variables in the steps are filled either by the user, such as the secret code, or by the system such as the name of the primary actor. When there are alternative paths available for a step, the user can select one from the contextual menu that displays the possible alternative as showed in the figure. The same idea applies to the specification of the repetition for the loops.


Error Log
Use Case Editor
Scenario Editor
Console
Rent a movie
Update parameters
Apply alternative Wrong secret code
Apply alternative Three times wrong code

No	Description	Use Case
1	Customer enters his secret code 1234	Authentication
2	System validates the secret code	Authentication
3	System shows a warning for wrong secret code	Authentication
4	Customer enters his secret code 1111	Authentication
5	System validates t	
6	The system says H	
7	Customer selects the movie Invictus	Rent a movie
8	Customer selects the movie Seven	Rent a movie
9	Customer validates his selection	Rent a movie
10	System validates card credit	Rent a movie
11	System gives selected movies	Rent a movie
12	System says Good bye !	Use automated DV...
13	System gives back the card	Use automated DV...

Figure 20. Scenario view

6. RELATED WORK

In the literature, we found the meta models of Somé [20] and Hoffmann et al. [11] to be the most elaborated. This is why we spend more time detailing their proposal than the others. Somé [20] concentrates on the modeling of the textual description of the use-case i.e. the description of the interactions between system and actors. In this work, the "behavior" is represented by a specialization of the UML metaclass *Behavior* which owns two specializations: *NormalDescription* that represents what he calls the "traditional use-case" and *ExtendDescription* to represent extension use-case. This distinction is necessary since Somé wants to closely follow the UML definition of extension use-case as sets of chunks of behavior, each chunk corresponding to an extension point of the target use-case. Therefore, the interactions are not modeled the same in both kind of use-cases. For *NormalDescription*, the interactions are represented as a collection of steps. Indeed, it owns a link to the *StepSequence* class which itself is associated to a collection of Step. The modeling of *NormalDescription* bears therefore some similarity with our own modeling. However, as explained above, we chose to stick to the definition of the extension use-cases as proposed by Jacobson, rather than adopting the UML's. This is why our work differs from the one of Somé. In our work, extended use-cases are full featured use-cases. In Somé's work, *ExtendDescription* is associated to a set of *Fragments* that are themselves associated to one *StepSequence* class which owns a collection of *Steps*. There must be as many fragments in an *ExtendDescription* as *ExtensionPoint* in the extended use-case. Finally, Somé does not model the scenarios nor the generalization relationship among use-cases. Hoffman et al. [11] called the textual specification of the behavior the *NarrativeDescription* since it is narrative in essence. *NarrativeDescription* has an association to the class *Flow* which itself is a collection of *Event. Events* are of two kinds: *Action and ContextSwitch*. The first represents an executable step, for example a user interaction with the system, while the second is a way to model the composition of flows i.e. the "spots in a flow where behavior of another flow can or must be inserted" [11]. *ContextSwitch* is further specialized in *Inclusion* and *ExtensionAnchor*. Both of them are further specialized in two classes to account for the internal or external source of flow to be inserted. However, the idea of *ExtensionAnchor* seems redundant with the *ExtensionPoint* of the UML meta model. Besides, Hoffman introduced a sophisticated concept to model the triggering and insertion conditions of a flow in another flow: the *Context*, which is associated to a flow. In particular, the latter is specialized as *InclusionContext* and *ExtensionContext*. These are associated to the corresponding specialization of *ContextSwitch* to model where the flows are inserted. But Hoffmann does not model the generalization relationship among use-cases. Although his formalization has its own merits it is very far from the original UML meta model. Moreover, the relationship of the Hoffman's model to the UML meta model is unclear to us since the many new concepts introduced in this work seem sometimes redundant with what is already represented in the UML meta model. In contrast our approach is to comply as much as possible with the UML meta model. Therefore the models of Hoffman at al. and ours are barely comparable. The work of Zelinka at al. [23] is much closer to ours since they explicitly modeled the *Flow*, the flow *Step* and reused these concepts to show where in a flow the insertion or extension flows must be inserted. However, the impact on the UML meta model of Zelinka's proposal is much bigger than ours. In fact they introduced two extra relations *FlowInclusion* and *FlowExtension* that link together a *Flow* i.e. a set of steps, an inclusion (extension step) and the *Include* (*Extend*) relationship. However we consider this modeling awkward since the UML *Include* (*Extend*) relationships do represent the inclusion (extension) flow since they relate two use-cases that themselves have flows. Finally Zelinka's work does not model the Generalization relationship nor the concept of a scenario. Nakatani et al. have taken another perspective, that of modeling the use-case behavior as activity diagrams [15]. Although their approaches include the definition of many perspectives under which the use-case can be modeled, we will only comment on the activity perspective since this is the way the behavior is defined. Under this perspective, the proposed meta model merges the Activity and Use Case UML meta model. However, the link between the *Include* or *Extend* UML relationships and their activity diagram's counterpart is not formally defined. Moreover they introduced the concept of *Composite Use Case* as a specialization of the UML's *Use Case* class without further explanation on the role of this class and especially its relationship to the *Include* or *Extend* relations. In parallel they defined the *Composite Activity* as a Specialization of the Activity class. We suspect this new class to be somewhat related to the *Composite Use Case* class, but this is not explained. All in all, many new elements have been defined in Nakatani's meta model whose theoretical justification is not given. Again, we suspect many of these new concepts to be redundant or in conflict with what is already modeled in the UML meta model. But this is hard to tell considering the scarce explanation provided on the model. The model proposed by Bragança and Machado [3] concentrates on the precise definition of the inclusion and extension points. As for the definition of the behavior, they rely on the Activity concept of UML. Therefore, inclusion and extension points refer to activities to identify the location where the new behavior should be inserted. But, as explained in the introduction, the way UML models the behavior of use-cases is not very detailed. As a consequence there does not seem to be any difference between alternative flows modeled "inside" a use-case and the Extension use-case. Bragança and Machado seem to have mixed both concepts. Like in the other articles reviewed, they do not model the generalization relationship nor the scenario. Finally, Rui and Butler [19] present an early work on the definition of flows in use-cases. In fact, the behavior of use-cases is defined as *Episodes* themselves made of *Events* that are further specialized in *Stimulus*, *Response* and *Action*. But this work does not formally define the include, extend and generalization relationships, nor the concept of alternative flow. However, the *Episode* class has a "consist-of" dependence to itself that we suspect could represent some composition constraint between *Episodes*. Finally the concept of scenario is not modeled in [19] either.

7. CONCLUSION

The key contribution of this paper is a precise yet simple modeling of all the relationship between the use-cases with minimal impact on the standard UML meta model. In particular, we modeled the Generalization relationship that we did not find anywhere else. Moreover we precisely defined the difference between use cases and scenarios and included the latter in the meta model. This seems to be unique to our work. Then we showed how the meta model can be extended to account for the association between the flow steps and the elements of the robustness diagram. This is the first time the UML specification and analysis models are formally linked together. From this background, we are building a reverse engineering environment were the traceability links between the source code and the use-cases can be reconstructed. Finally we presented our use-case and scenario editing tool that is built on top of the proposed meta model. In fact, we relied on EMF /GMF to actually generate the Eclipse editors from our meta models. This represents the ultimate check for meta model completeness. This guarantees that the meta model complies with all the needed modeling requirements and constraints. The next step in this project is to implement an inference engine to exploit the use-case and analysis model's information to automate the reconstruction of the traceability links between the source code and the specifications. The current work deals with the definition and implementation of the inference rules and approximate reasoning technique to let us model the source-code to analysis model mapping heuristics.

8. ACKNOWLEDGMENTS

We gratefully acknowledge the financial support from the Swiss Confederation's KTI/CTI, grant N° 10448.1 PFES-ES.

9. REFERENCES

[1] Ambler S.W. 2002. *Agile Modeling*. John Wiley and Sons. New-York.

[2] Ambler S.W. 2004. *The Object Primer*. Cambridge University Press; 3rd edition. Cambridge UK.

[3] Bragança A., Machado R.J. 2006. Extending UML 2.0 Metamodel for Complementary Usages of the «extend» Relationship within Use Case Variability Specification. *Proc. of the 10th IEEE Int. Software Product Line Conf.*

[4] Bittner, K. and Spence, I. 2002. *Use Case Modeling*. Pearsons Education Inc. Boston.

[5] Cockburn, A. 2001. *Writing Effective Use Cases*. Addison-Wesley. Reading, Massachusetts.

[6] Cockburn, A. 2002. *Use cases, ten years later*, http://alistair.cockburn.us/Use+cases%2c+ten+years+later. Accessed on August 11th, 2010.

[7] Dugerdil, P. 2006. Reengineering Process Based on the Unified Process. *Proc. of the 22nd IEEE Int. Conf. on Software Maintenance (ICSM). 2006.*

[8] Dugerdil, P. and Jossi, S. 2007. Reverse-engineering of an industrial software using the unified process: an experiment. *Proc. of the 11th IASTED Int. Conf. on Software Engineering and Applications.*

[9] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/

[10] Graphical Modeling Project. http://www.eclipse.org/modeling/gmp/

[11] Hoffmann, V., Lichter, H., Nyßen, A. and Walter, A. 2009. Towards the Integration of UML- and textual Use Case Modeling. *J. of Object Technology*, 8(3), May-June 2009.

[12] IBM Rational Unified Process. http://www-01.ibm.com/software/awdtools/rup/

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

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

[15] Nakatani T., Urai T., Ohmura S., Tamai T. 2001. A Requirements Description Metamodel for Use Cases. *Proc of the 8th IEEE Asia-Pacific on Software Engineering Conference, 2001.*

[16] OMG. Meta-Object Facility. http://www.omg.org/mof/

[17] OMG. UML Superstructure Specification, Version 2.3. http://www.omg.org, May 2010.

[18] Rosenberg D., Stephens M. 2007. *Use Case Driven Object Modeling with UML, Theory and Practice*. Springer-Verlag Inc. New-York..

[19] Rui K., Butler G. 2003. Refactoring Use Case Models – The Meta Model. *Proc. 26th Australasian Comp. Science Conference.* Adelaide, South Australia

[20] Somé, S. 2009. A Meta-Model for Textual Use Case Description. *J. of Object Technology*, 8(7), Nov.-Dec. 2009.

[21] Whittle J., Jayaraman P.K. 2006. Generating Hierarchical State Machines from Use Case Charts. *Proc 14th IEEE Int. Requirements Engineering Conference (RE'06)*

[22] Wirfs-Brock R., McKean A. 2003. *Object Design*. Addison-Wesley. Reading, Massachusetts.

[23] Zelinka L.,Vranic V. 2009. A Configurable UML Based Use Case Modeling Metamodel. *Proc of the First IEEE Eastern European Conf. on the Engineering of Computer Based Systems.*