JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 3, May-June 2009

Towards the Integration of UML- and textual Use Case Modeling

Veit Hoffmann, Horst Lichter, Alexander Nyßen and Andreas Walter
RWTH Aachen University, Germany

Abstract

In this paper, we present a metamodel for textual use case descriptions, structurally conforming to the UML, to specify the behavior of use cases in a flow-oriented manner. While being primarily targeted at supporting requirements engineers in creating consistent use case models, the metamodel defines a textual representation of use case behavior that is easily understandable for readers, who are unaware of the underlying metamodel. Hence, the known benefits of natural language use case descriptions are preserved. Being formalized, consistency between UML-based use case representations and their textual descriptions can be automatically ensured. With NaUTiluS we present an extensible, Eclipse-based toolkit, which offers integrated UML use case modeling support, as well as editing capabilities for their textual descriptions.

1 INTRODUCTION

Since their invention by Ivar Jacobson in 1986 [Jacobson87, Jacobson04], and although having some deficiencies [Glinz00, Williams05] use cases have gained wide-spread acceptance as a means to describe interactions between a system and its environment [McPhee02, Neill03]. Today, the Unified Modeling Language [OMG07] is a widely accepted standard defining the central use case modeling concepts. There is a manifold of notations to describe the behavior captured by use cases in detail. While the UML offers various diagram types (state machine, sequence, activity) to describe internal behavior, textual descriptions, e.g. proposed in [Rolland98, Cockburn00, Li00], which are most widely used, are not addressed.

As a consequence, when applying a UML-based development approach, use cases are first identified and structured by means of UML use case diagrams and then described in detail through textual descriptions. Thus, the so-called *use case model* is actually a composite model consisting of two parts. One part is a UML model, capturing the use cases and their relationships, the other part is a set of textual descriptions of the behavior represented by these use cases. These two parts depict different views on the overall use case model and should of course not contradict each other.

Since the two views evolve in parallel during the process of use case modeling, ensuring consistency between them is an ongoing task. In order to provide automated consistency checking between a UML use case model and its corresponding set of textual descriptions, the textual representations of the use case relationships contained in the UML model must be algorithmically and efficiently identifiable. This is a non-trivial problem, because ensuring consistency between a UML model and the textual use case descriptions requires a certain degree of formality in the textual descriptions. On the other hand the benefit of use case modeling is mainly rooted in its semi-formal nature. For this reason, a new format of textual use case descriptions has to be defined, which is easily understandable for a human reader but at the same time facilitates efficient algorithmic identification of the aforementioned correspondence between a UML use case model and the set of textual descriptions.

The rest of this paper is organized as follows. In the next section we shortly introduce a format for textual use case descriptions, which provides the conceptual background for the approach presented here. Then we present in section 3 a complete metamodel for textual use case descriptions, the so-called *narrative metamodel* and its integration into the UML metamodel. In section 4 we briefly introduce NaUTiluS, a use case modeling tool that implements both, the UML use case metamodel and the narrative metamodel. Finally, we summarize the approach presented in this paper and give an outlook towards further research and ideas concerning the application of the narrative metamodel.

2 RELATED WORK

Interpretations of what a use case actually is differ, and so do the notations provided to describe use cases in a detailed manner. There are many published approaches presenting formal notations to capture use case behavior. Very often the different behavioral UML diagrams (e.g. state machine, sequence, or activity diagrams) are used [Kholkar05, Whittle06]. But also colored petri-nets [Jorgenson04] or even formal specification languages like Z [Spivey92] are proposed. As all of these notations are based on a defined formalism, they provide the possibility to inspect and analyze the use case descriptions automatically and to use these formal models in succeeding activities (e.g. test case design [Reuys06, Ryser00]). However the major drawback of using formal notations is rooted in their formality as well, since formal descriptions are difficult to understand by non-technical stakeholders. Therefore readability and understandability are key requirements for use case description notations.

Because of that we focus our interest on notations that allow a human reader with no or only little formal background to quickly understand use case descriptions. While the often applied approach to describe use case behavior in a template-based textual form (proposed e.g. by Cockburn [Cockburn00], Armour and Miller [Armour01] or Kulak and Guiney [Kulak03]) leads to readable and understandable descriptions, those descriptions cannot be analyzed or validated automatically.

On the other side, all approaches proposing the usage of a constraint form of natural language to describe use case details in a way that formal models can be generated by means of language analysis [Li00, Drazan07], have drawbacks concerning their applicability and implementability.

A very promising approach, which introduces some degree of formality into use case descriptions while allowing unconstraint natural language, was presented by Bittner and Spence [Bittner03]. Bittner and Spence describe use case details in a flow-oriented way, which is well aligned with current UML definitions and results in descriptions that are easy understandable for non-professionals as well. Our metamodel is based on their basic ideas and we explain this flow-oriented behavior description in the following section.

Flow-oriented Use Case Description

According to the UML, a use case represents a variety of scenarios that can result from the interaction between a system and its environment. Its description has to cover all possible scenarios. Since it is hard and often impossible to name and describe each of them in isolation, a use case is often described in an *incremental* fashion: One scenario is described completely and explicitly, and all other scenarios are described implicitly in terms of their differences to the first one.

A variant of this pattern is chosen in [Bittner03], where partitioning the description of the behavioral spectrum represented by a use case into so-called *flows of events* is proposed. The *basic flow* describes a sequence of events occurring under some conditions regarded as the "default case". This sequence depicts one possible course through the execution of the use case and serves as a starting point for its description.

In order to specify possible variants of the behavior described by the basic flow, it may contain so-called *extension points*. Each extension point is a named placeholder representing possible behavioral variations to occur under defined conditions. Such variations are described in terms of *alternative flows*, which themselves are sequences of events. The concept of variation by alternative flows can be applied recursively, thus allowing the description of arbitrarily complex behavioral spectra.

In [Bittner03], extension points are not only used to describe varying behavior assigned to one use case, but also to establish extension relationships between use cases in their textual description. To this sense, some extension points contained in a use case description are denoted as "public", thus allowing to be referenced from other textual descriptions. This way, the description of an extending use case can reference extension points in the description of an extended use case to represent the *extend* relationship between use cases, as defined by the UML.

Generalization between use cases is represented similarly by referencing selected extension points within the description of the general use case and providing the behavioral differences that account for the specificity of the specialized use case.

3 A METAMODEL FOR TEXTUAL DESCRIPTIONS

Goals

With the narrative metamodel we present here, we pursue three major goals.

- First, the textual use case descriptions should capture the specific nature of use cases, i.e. they represent all possible scenarios from which one occurs during each of a use case's executions.
- Second, the textual use case descriptions should be written in unconstrained natural language, because we do not want to force the writer and/or reader to learn a complex notation or grammar.
- Third, the consistency between a set of textual use case descriptions and the UML use case model, defining these use cases, should be ensured automatically. This especially includes checking whether all use case elements, expressed in the UML model, are also represented consistently in the textual descriptions.

UML-based Use Case Models

To have a notion of consistency between a UML model and a corresponding set of textual descriptions, we first depict which elements constitute the UML model and thus have to be represented in the textual use case descriptions.

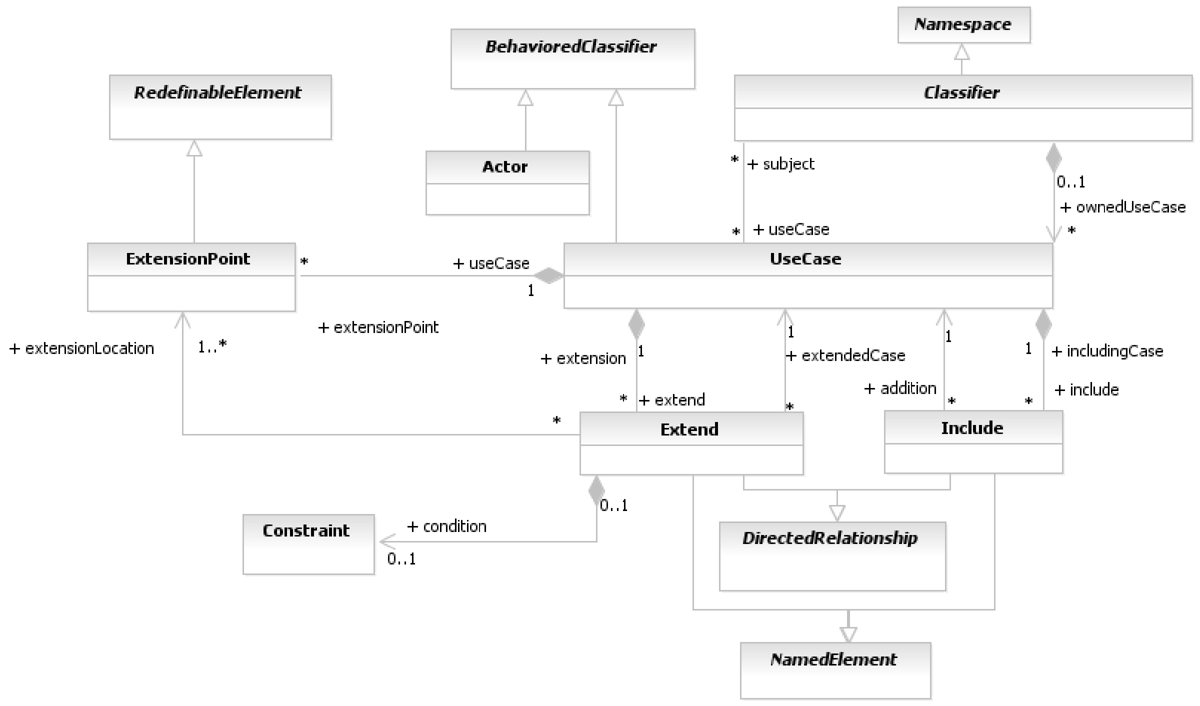


Figure 1: The metaclasses defined in the Language Unit *Use Cases*

The kinds of elements constituting a UML use case model are primarily defined by the metaclasses contained in the UML Language Unit *Use Cases* [OMG07], whose class diagram is shown in Figure 1. Obviously important metaclasses are *UseCase*, *Actor*, *Include*, *Extend* and *ExtensionPoint*. The metaclass *Generalization* is not shown in the diagram because it belongs to the UML Language Unit *Classes*, the same holds for the metaclass *Association* from UML Language Unit *Kernel*.

A UML model is usually structured as a composition hierarchy, where every element except the root element, which is usually a *Model*, is contained by another model element. In the context of use case modeling, *Includes*, *Extends*, *ExtensionPoints* and *Generalizations* are usually directly contained by a *UseCase* or, in the case of *Generalization*, also by an *Actor*. Furthermore, *Package*s are used to group logically related model elements. Being a *PackageableElement*, a use case as well as an actor can be contained in a package. Use cases can alternatively be contained in a *Classifier* which denotes the subject a use case applies to, i.e., the system that is described. Instead of allowing instances of any concrete subclass of *Classifier* to represent the described system, *Components* are usually used, and we define this to be the only legal case. This allows to group all use cases of one specific subject (the modeled system) semantically and visually, i.e. via diagrams, in several packages contained in the corresponding component.

Running Example

In the following we introduce the narrative metamodel and its concepts alongside a brief example, which eases understanding the metamodel elements and their roles in the textual use case descriptions.

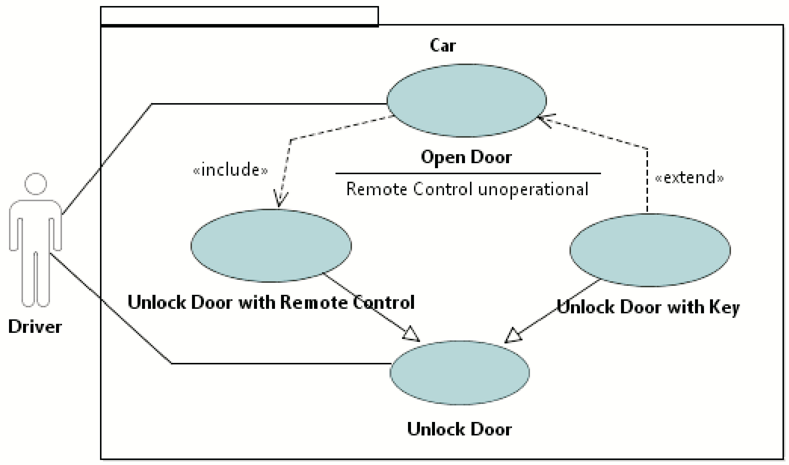


Figure 2: A UML use case diagram defining the “Open Door” use cases

The example depicts a UML use case model for opening the doors of a car either by means of a remote control (this is considered to be the default way) or with a key. While Figure 2 shows the UML use case model, Figure 3 shows the corresponding flow-oriented textual use case descriptions conformant to our narrative metamodel.

The description of the use case **Open Door** defines besides its main flow the flow **Switch off Alarm** describing exceptional behavior that may be performed at any time during the execution of the main flow. Afterwards the flow continues at the point of the execution, where the exceptional behavior has been triggered.

The use case **Unlock Door with Key** contains the alternative flow **Unlock only one Door with Key**. This flow encapsulates behavior that may be performed alternatively to the default behavior, described in the main flow of use case **Unlock Door with Key**.

UC Open Door
Main Flow:
Contexts:
1. *Is invoked by Actor (Driver)*

Events:
1. *The driver approaches the car*
2. *Include UC Unlock with Remote Control to unlock the car's doors*
3. *The driver checks if the doors are unlocked*
4. *{Remote Control unoperational}*
5. *The driver pulls the handle and opens the door*

Exception Flow (Switch off Alarm):
Contexts:
1. *At any time in UC Open Door (Main Flow) if alarm raised*

Events:
1. *The driver switches off the alarm*

UC Unlock Door with Remote Control
Main Flow (redefines UC Unlock Door Main Flow):
Contexts:
1. *Is invoked by Actor (Driver) (inherited from UC Unlock Door Main Flow)*
2. *Is included by UC Open Door (Main Flow)*

Events:
1. *The driver unlocks the car with the remote control (redefines UC Unlock Door: The driver unlocks the car)*

UC Unlock Door (abstract use case)
Main Flow:
Contexts:
1. *Is invoked by Actor (Driver)*

Events:
1. *The driver unlocks the car*

UC Unlock Door with Key
Main Flow (redefines UC Unlock Door Main Flow)
Contexts:
1. *Is invoked by Actor (Driver) (inherited from UC Unlock Door Main Flow)*
2. *Extends UC Open Door at {Remote Control unoperational} if Remote Control is unoperational*

Events:
1. *{No central locking system}*
2. *The driver unlocks the car with the key {End Main Flow}*

Alternative Flow (Unlock only one Door with Key):
Contexts:
1. *At {No central locking system} if car has no central locking system*

Events:
1. *The driver selects a door to unlock*
2. *The driver unlocks the selected door with the key*

Resume Unlock with Key Main Flow at {End Main Flow}

Figure 3: The "Open Door" textual use case descriptions

4 THE NARRATIVE METAMODEL

Integration with the UML Metamodel

Since a textual description of a use case is to a certain degree narrative, we consequently call it a *NarrativeDescription* (see Figure 4). To be able to represent relationships between actors and narrative descriptions of use cases, all actors of the UML use case model are represented by respective *NarrativeActors*.

Since it is recommendable to group narrative descriptions the same way as the corresponding use cases in the referenced UML model (see Figure 4), the metaclass *NarrativeContainer* defines a correspondence to the UML “container classes” *Package* and *Component*, thus creating a hierarchy. As a correspondence to the UML metaclass *Model*, the metaclass *NarrativeModel* defines the root element of the *NarrativeContainer* hierarchy.

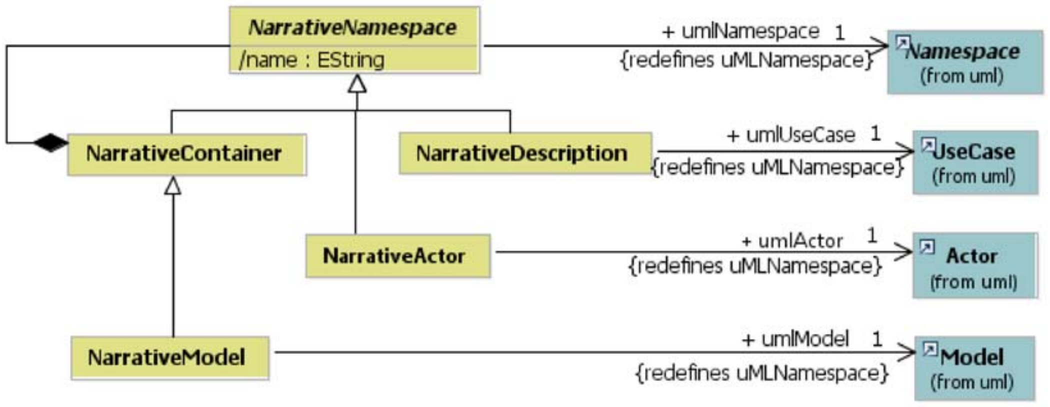


Figure 4: Core classes of the narrative metamodel

By means of instances of the metaclasses presented so far, a coarse-structural synchronization between a UML model and a narrative model is established. Having a one-to-one correspondence between use cases in the UML model and narrative descriptions in the narrative model, it is possible to describe each use case in detail and to unambiguously represent its properties captured in the UML model in the narrative model, too. This is covered in the following sections.

Flows and Events

Following the idea of Bittner and Spence [Bittner03], the description of a use case is split into so-called *Flows* which define behavioral fragments through sequences of *Events*. We depict those concepts with respective metaclasses as well (see Figure 5).

In the most simple case, the events of a flow are atomic *Actions* whose content is not interpreted from the point of view of the metamodel. This means that the writer can use unconstrained natural language to describe those basic behavioral elements.

All other kinds of events symbolize spots in a flow where behavior of another flow can or must be inserted and thus, the context may change (see Section "Contexts"). Therefore a second kind of event is introduced by the metaclass *ContextSwitch*, which represents all concepts where changes of the current context can occur. We distinguish two kinds of context switches which are again specialized.

First a flow may include another flow in its execution. This relationship is formalized by the metaclass *Inclusion*, which is further refined into two variants. The first variant, *InternalInclusion*, represents the inclusion of a flow being contained in the same narrative description, a so-called subflow. The second, *ExternalInclusion*, represents the inclusion of a flow residing in another narrative description, which corresponds to the UML include-relationship between use cases.

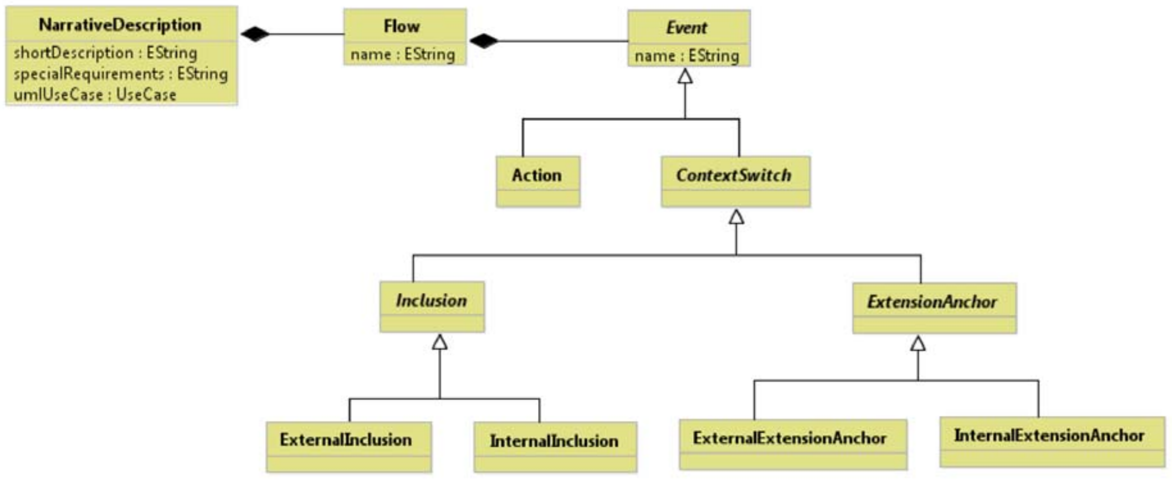


Figure 5: Metaclasses representing the main concepts of a narrative description

Since a use case usually represents a variety of interaction sequences, from which one is followed during its execution, it must be possible to describe behavioral variations occurring under different conditions. Therefore we introduce a second kind of context switch, *ExtensionAnchor*, which marks a point where variation of the behavior described by a flow can occur. These extension anchors correspond to the extension points defined by Bittner and Spence [Bittner03], which were briefly described in section 2. Unlike inclusion relationships, where the including flow defines the point where the additional behavior will be included, an extension anchor merely expresses that a variation of the described behavior is possible. However, how and under which conditions the behavior is varied, is specified by the extending flow.

UC Open Door
Main Flow:
Contexts: ...
Events:

1. *The driver approaches the car*	← Action
2. *Include UC Unlock with Remote Control to unlock the car's doors*	← ExternalInclusion
3. *The driver checks if the doors are unlocked*	← Action
4. *{Remote Control unoperational}*	← ExternalExtensionAnchor
5. *The driver pulls the handle and opens the door*	← Action

Figure 6: A flow consisting of different kinds of events

In this context, two different scenarios are possible. First variational behavior can always be contained within the same narrative description. In this case, we are not restricted to the description of additional behavior possibly occurring at specific points in the flow, like defined by the UML, but we can describe more complex variations. In any case, whenever a variation of the behavior described by a flow is assigned to the same narrative description, we use an *Internal-ExtensionAnchor* (see Figure 5) to mark the point, where this variation can occur.

Second, if additional behavior can occur at one or more defined points in the flow and never ends or bypasses but only extends the varied behavior in the literal sense, as defined by the UML, the additional behavior can be represented by a second use case that extends the first one. We model such points in the flow, where its behavior can be extended by another flow of a different narrative description, by *ExternalExtensionAnchors* (see Figure 5). They correspond to *ExtensionAnchors* in the UML model.

Figure 6 illustrates how the concepts described in this section can be applied to our running example. The main flow of the use case **Open Door** consists of five distinct events that describe the use case's behavior (three atomic actions, one external inclusion, one external extension anchor).

Contexts

While a context switch specifies where the behavior defined by a flow is inserted from the viewpoint of the inserting flow, *Contexts* describe the same concept from the viewpoint of the included respectively extending flow.

We distinguish four kinds of contexts needed to initiate the execution of the behavior described by a flow: *InteractionContexts*, *InclusionContexts*, *ExtensionContexts* and *Exception-Contexts* (see Figure 7).

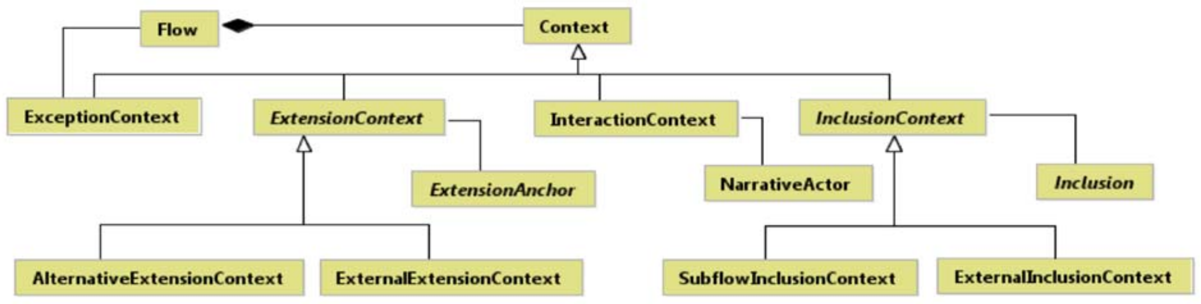


Figure 7: Metaclasses representing different kinds of contexts

Whenever a flow is triggered directly by a narrative actor, thus having a primary actor, the flow must have an *InteractionContext* with this narrative actor. This special context defines the conditions under which the behavior encapsulated in the flow is triggered. These conditions have to be fulfilled before, possibly during, and at the end of the execution of the flow's behavior. In our example the main flow of the use case **Unlock Door with Key** has an *InteractionContext* with the narrative actor **Driver** (see Figure 8)

Besides being triggered by a narrative actor, a flow can be, as mentioned before, included by another. A flow that is included by a flow within the same narrative description, is assigned an *InternalInclusionContext*, a flow that is included by a flow belonging to a different narrative description is consequently assigned an *ExternalInclusionContext*. *ExtensionContexts* define the conditions under which a flow adds behavior to another flow at one or more extension anchors. Again we distinguish two different kinds of extension contexts: *ExternalExtension-Contexts* and *InternalExtensionContexts*. As explained before, an external extension anchor represents a UML extension point. A flow, which describes behavior that can occur at this point, has an *ExternalExtensionContext* that represents the UML extend-relationship. The condition for the extension to take place is thus already specified in the corresponding UML model (as an instance of *Constraint*, see Figure 1). An example for an external extension context is shown in Figure 8. The main flow of use case **Unlock Door with Key** has an external extension context respective to the main flow of use case **Open Door**. Again, unconstrained natural language can be used to specify the conditions, under which the respective behavior is inserted, because we do not want to formalize conditions too much.

UC Unlock Door with Key
Main Flow (redefines UC Unlock Door Main Flow):
Contexts:
1. *Is invoked by Actor (Driver)* ← InteractionContext
   (inherited from UC Unlock Door Main Flow)
2. *Extends UC Open Door at* ← ExtensionContext
   {Remote Control unoperational}
   if Remote Control is unoperational

Events: ...

Figure 8: A textual description with different kinds of contexts

Since the narrative metamodel allows to describe complex behavioral variations, e.g. a flow can bypass another flow's behavior, it must also be explicitly specified where a flow's execution is continued when the exceptional behavior ends. Again, a set of *InternalExtensionAnchors* and corresponding conditions can be specified to denote the options to continue the flow's execution.

At last, if a flow has an *ExceptionContext*, its behavior can be inserted into another flow at any time when the defined condition of the exception context is met. Besides that, the exception context specifies, where the execution of the suspended flow continues under different conditions. This can in contrast to [Metz03] either be the event, where the exceptional condition was met, the end of the current flow, or the end of the whole scenario. In our running example (see Figure 3), the alarm can be raised at any time, and whenever it is raised, the behavior defined in the use case **Switch off Alarm** is executed. Afterwards the execution of the main flow is continued after the event, where the exception condition was first met.

Each flow of a narrative description can have many different contexts. A flow can for example be associated directly to a primary actor, thus having an interaction context. Furthermore, it can be included in another flow of a different narrative description and thus has an inclusion context, too. There are several restrictions to the kinds of different contexts a flow can have, e.g. only one flow in a narrative description may have inclusion contexts and/or interaction contexts. Due to space limitations we can not explain these restrictions but all of them are included in the metamodel as OCL constraints (see Section "Enhancing Model Quality through Constraints").

Representing Generalization between Use Cases

The concepts presented so far allow modeling a behavioral spectrum, from which one concrete occurrence is determined *dynamically* based on the evaluation of the specified conditions during a use case's execution.

With its *Generalization* relationship, the UML allows to model that behavior represented by one use case is a specialization of the behavior represented by its general use case. This relationship between use cases is defined statically, i.e., on the model level. In our metamodel, a generalization relationship between use cases that is captured in a UML model is reflected by a derived association between the related narrative descriptions respectively, thus being directly inferred from the generalization hierarchy of the corresponding use cases in the UML model (see Figure 9).

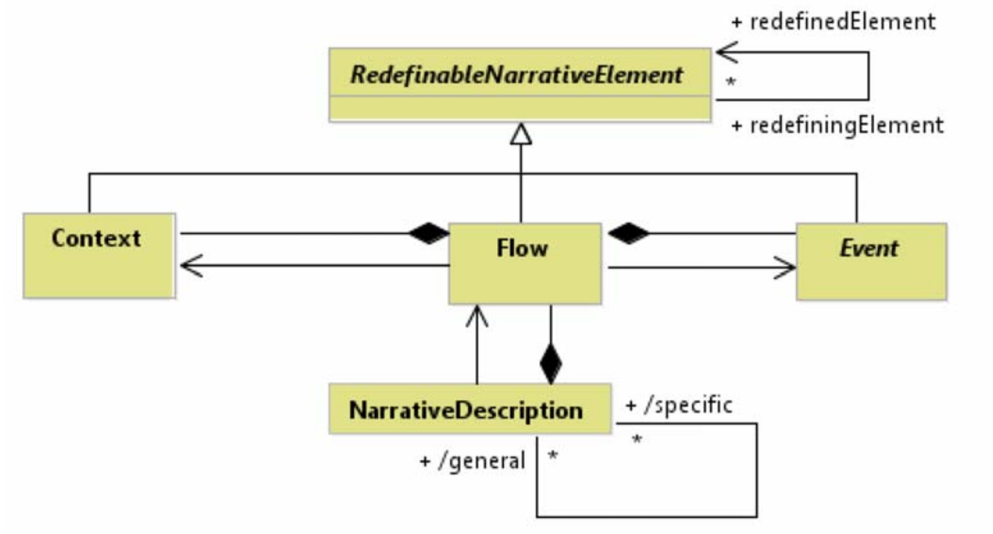


Figure 9: Generalization and redefinition of narrative descriptions

If a narrative description is a specialization of another one, the specific narrative description inherits all flows of the general one (with their events and contexts). In order to represent the specific narrative description's nature, flows, events, and contexts of the general narrative description can be redefined. We consequently call these elements *RedefinableNarrativeElements* (see Figure 9).

The redefinition of narrative elements is realized in two steps within a specialized narrative description. First, the flows of the general narrative description can be redefined within the specialized one. Second, the elements of flows, events, and contexts can be redefined separately within a redefined flow. Thus a flow, redefining a flow of a general narrative description doesn't mask the redefined flow completely, but initially inherits all elements of the flow it redefines. Within the redefined flow the inherited contexts and events can be redefined explicitly to replace the inherited ones. Besides that, new contexts and events can be introduced, thus enhancing the described behavior. Figure 10 shows exemplarily how the main flow of use case **Unlock Door** is redefined in the specialized use case **Unlock Door with Remote Control**.

UC Unlock Door (abstract use case)	← general Description
Main Flow:	← general Flow
Contexts:	
1. Is invoked by Actor (Driver)	
Events:	
1. The driver unlocks the car	
UC Unlock Door with Remote Control	← redefined Description
Main Flow (redefines UC Unlock Door Main Flow):	← redefined Flow
Contexts:	
1. Is invoked by Actor (Driver) (inherited from UC Unlock Door Main Flow)	← inherited from UC Unlock Doors Main Flow
2. Is included by UC Open Door (Main Flow)	← defined locally
Events:	
1. The driver unlocks the car with the remote control (redefines UC Unlock Door The driver unlocks the car)	← redefined Action

Figure 10: An example depicting the redefinition of flows and events

Enhancing Model Quality through Constraints

In order to ensure the consistency between a UML use case model and a narrative model and to control the sanity of the flows inside the narrative descriptions, the metamodel has been enriched with a set of OCL [OMG06] constraints. We won't explain the single constraints in detail here, but we will give a brief overview on the goals and benefits we achieve through model constraints. Besides those constraints defined in the UML Use Case Language specification [OMG07], we distinguish two types of model constraints: *consistency constraints* and *sanity constraints*.

- Consistency constraints guarantee, that instances of the narrative metamodel are consistent with their adjacent UML use case model.
- Sanity constraints are by far the most important constraints we introduced, since those constraints define the exact semantics of the elements of the narrative metamodel. A violation of one of those constraints e.g. indicates that a narrative model has missing or incorrect flow definitions.

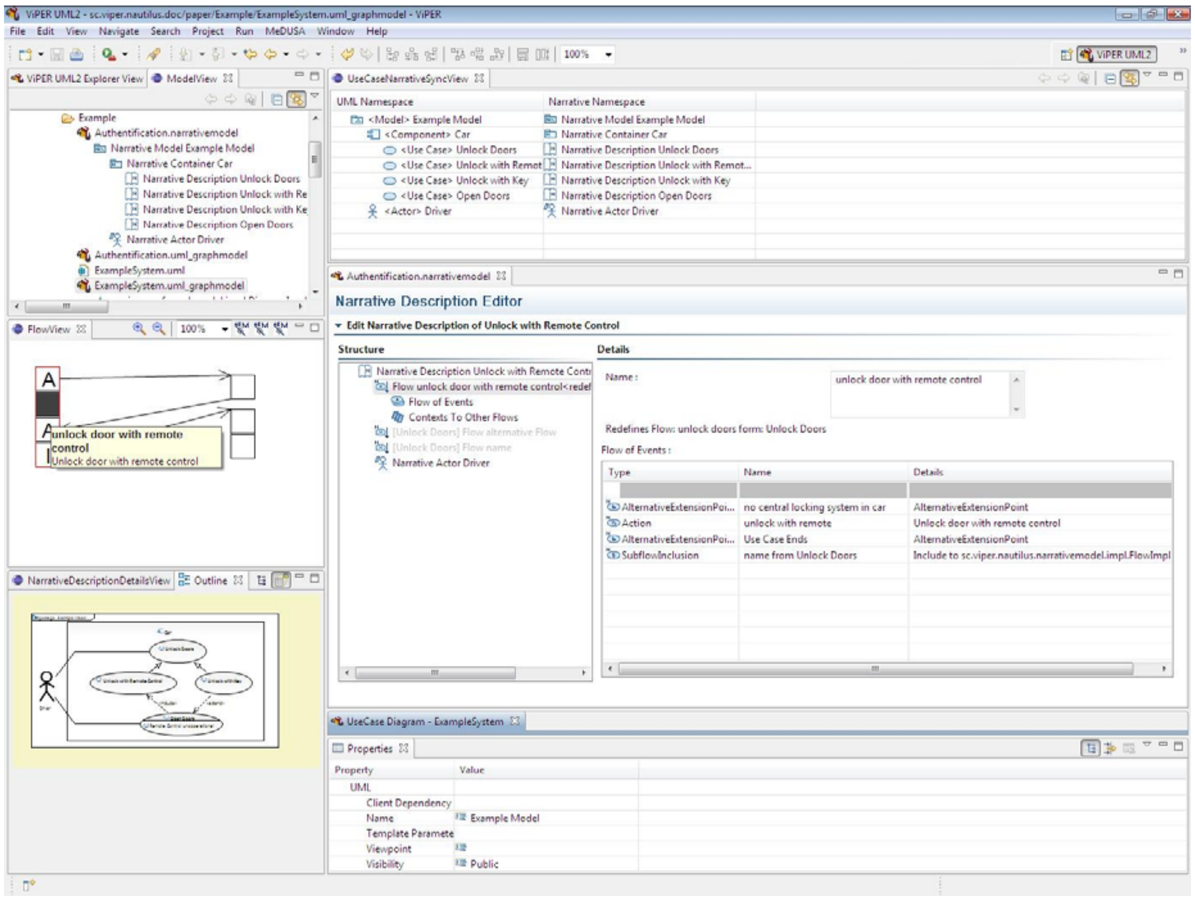
By means of the defined constraints it is guaranteed that in a model without any constraint violations all references between model elements are established and that each possible scenario of the narrative model has a valid start point and is described by an unambiguous, finite sequence of events. Thus we can call such a model *correct* in the sense that all its contained flows are linked correctly and all elements of the adjacent UML model are represented validly. What we obviously cannot determine is whether the use case descriptions are complete in the sense that all alternative flows are defined or that the narrative descriptions actually specify the actor-system-interactions correctly.

5 A NARRATIVE USE CASE EDITOR

To support a requirements engineer in developing the complete use case model and to ensure consistency between the textual use case descriptions and the corresponding UML model as well as sanity of the textual descriptions themselves, appropriate tool support is essential.

Most existing tools that support textual use case descriptions have major drawbacks. While UML tools like *Rational Software Architect* [Rational] or *EclipseUML* [Omondo] do not support the description of the internal structure of use cases, tools focusing on textual use case descriptions like *CaseComplete* [CaseComplete] or *UCEd* [UCED] leave out the use case constructs defined by the UML specification. Besides, none of the existing tools provides support to control the consistency between a UML model and its adjacent textual descriptions, since none of the textual descriptions is based on a formal metamodel.

To overcome these deficiencies, *NaUTiluS* (**Na**rrative **U**se Case Description **T**oolk**i**t for Eva**lu**ation and **S**imulation), a toolkit for narrative use case modeling, has been prototypically implemented. *NaUTiluS* consists of a set of plug-ins that are embedded in the *ViPER* platform [Viper]. It implements a couple of views and editors that support inspection and editing of *NarrativeModels*, as well as UML models (diagrams).


ViPER UML2 - sc.viper.nautilus.doc/paper/Example/ExampleSystem.uml_graphmodel - ViPER
File Edit View Navigate Search Project Run MeDUSA Window Help
100%
ViPER UML2
ViPER UML2 Explorer View
ModelView
UseCaseNarrativeSyncView
FlowView
NarrativeDescriptionDetailsView
Outline
Authentification.narrativemodel
UseCase Diagram - ExampleSystem
Properties
A
unlock door with remote control
Unlock door with remote control

Example
Authentification.narrativemodel
Narrative Model Example Model
Narrative Container Car
Narrative Description Unlock Doors
Narrative Description Unlock with Re
Narrative Description Unlock with Ke
Narrative Description Open Doors
Narrative Actor Driver
Authentification.uml_graphmodel
ExampleSystem.uml
ExampleSystem.uml_graphmodel

UML Namespace	Narrative Namespace
<Model> Example Model	Narrative Model Example Model
<Component> Car	Narrative Container Car
<Use Case> Unlock Doors	Narrative Description Unlock Doors
<Use Case> Unlock with Remot	Narrative Description Unlock with Remot...
<Use Case> Unlock with Key	Narrative Description Unlock with Key
<Use Case> Open Doors	Narrative Description Open Doors
<Actor> Driver	Narrative Actor Driver

Narrative Description Editor

Edit Narrative Description of Unlock with Remote Control

Structure

Narrative Description Unlock with Remote Contr
Flow unlock door with remote control<redef
Flow of Events
Contexts To Other Flows
[Unlock Doors] Flow alternative Flow
[Unlock Doors] Flow name
Narrative Actor Driver

Details

Name : unlock door with remote control

Redefines Flow: unlock doors form: Unlock Doors

Flow of Events :

Type	Name	Details
AlternativeExtensionPoi...	no central locking system in car	AlternativeExtensionPoint
Action	unlock with remote	Unlock door with remote control
AlternativeExtensionPoi...	Use Case Ends	AlternativeExtensionPoint
SubflowInclusion	name from Unlock Doors	Include to sc.viper.nautilus.narrativemodel.impl.FlowImpl

Property	Value
UML	
Client Dependency	
Name	Example Model
Template Paramete	
Viewpoint	
Visibility	Public

Figure 11: The NaUTiluS Toolkit

Figure 11 shows a screenshot of *NaUTiluS* and some of its views presenting different aspects and information of the use case model. *NaUTiluS* supports creation, deletion and editing of all elements defined by the narrative metamodel. The toolkit automatically ensures the consistency of the textual descriptions and the UML model and is able to synchronize a narrative model with its adjacent UML use case model automatically.

Besides that, *NaUTiluS* validates the constraints we introduced (see Section "Enhancing Model Quality through Constraints"), to eliminate ambiguities and defects in the textual descriptions. Apart from simple editing operations on model elements, *NaUTiluS* offers features to refactor use case models (e.g. events can be moved between flows or refactored into a new subflow). To provide an easy way to integrate narrative models into a textual requirements specification, *NaUTiluS* currently offers a simple text export functionality. The exported text is formatted like the textual descriptions shown in the running example (compare e.g. Figure 3).

6 CONCLUSION AND OUTLOOK

Use case descriptions based to the presented narrative metamodel have a degree of formality needed to ensure consistency between the graphical UML use case model and its narrative descriptions algorithmically, without significant drawbacks to the readability and understandability of the textual descriptions.

First practical experiences show that textual descriptions conforming to the narrative metamodel can express the behavior encapsulated in a use case model well. The concept of flows-of-events is easily understandable even for stakeholders without expertise in use case modeling techniques. Especially the validation constraints, which were added to the narrative metamodel, help to identify inconsistencies quickly, thereby leading to an enhanced model quality.

NaUTiluS supports the editing of narrative models quite well. The offered views on the use case model help to understand existing models quickly and to keep track of the model elements during the creation and editing of a use case model. The offered feature to export a textual representation of the narrative model is a prerequisite to improve acceptance of the tool as well as to simplify the integration of a use case model into supplementary requirements documentation.

Currently we are enhancing *NaUTiluS* by a simulation environment. It will provide a feature to step through a narrative model and thus simulate the execution of the use case descriptions. The simulation of narrative models will enable the user to understand the described behavior quickly and better. Simulation will thus support the validation of the model and help to identify failures and hotspots in the modeled behavior quickly. Besides that, several analyses based on the dynamic structure of the simulated scenarios are conceivable. In the future we will provide capabilities for those enhanced model analyses.

Furthermore we will develop *NaUTiluS* support for evaluating the quality of use case descriptions by means of quality metrics based on the static structure of a narrative model as well as on its dynamic behavior.

REFERENCES

[Armour01] F. Armour and G. Miller. Advanced Use Case Modeling Volume One, Software Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Bittner03] K. Bittner and I. Spence. Use Case Modeling. Addison-Wesley, 2003.

[CaseComplete] CaseComplete. http://www.casecomplete.com/.

[Cockburn00] A. Cockburn. Writing Effective Use Cases. Addison-Wesley, 2000.

[Drazan07] J. Drazan and V. Mencl. Improved processing of textual use cases: Deriving behavior specifications. In Proceedings of SOFSEM 2007 LNCS 4362, pages 856–86, Harrachov, Czech Republic, January 20 - 26, 2007.

[Glinz00] M. Glinz. Problems and Deficiencies of UML as a Requirements Specification Language. In IWSSD '00: Proceedings of the 10th International Workshop on Software Specification and Design, pages 11–22, Washington, DC, USA, 2000. IEEE Computer Society, 2000.

[Jacobson87] I. Jacobson. Object-oriented development in an industrial environment. In OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications, pages 183–191, New York, NY, USA, 1987. ACM Press, 1987.

[Jacobson04] I. Jacobson. Use cases - Yesterday, today, and tomorrow. Software and System Modeling, vol 3(3):210–220, 2004.

[Jorgensen04] J. B. Jorgensen and C. Bossen. Executable Use Cases: Requirements for a Pervasive Health Care System. IEEE Software, vol 21(2):34–41, 2004.

[Kholkar05] D. Kholkar, G. M. Krishna, U. Shrotri, and R. Venkatesh. Visual specification and analysis of use cases. In SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization, pages 77–85, New York, NY, USA, ACM, 2005.

[Kulak03] D. Kulak and E. Guiney. Use Cases: Requirements in Context. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Li00] L. Li. Translating use cases to sequence diagrams. In Proc. of Fifteenth IEEE International Conference on Automated Software Engineering, pages 293–296, Grenoble, France, 2000.

[McPhee02] C. McPhee and A. Eberlein. Requirements engineering for time-to-market projects. In Proc. of the 9th IEEE International Conference on Engineering of Computer-Based Systems, page 17, Washington, DC, IEEE Computer Society, 2002.

[Metz03] P. Metz, J. O'Brien, and W. Weber. Specifying use case interaction: Types of alternative courses. Journal of Object Technology, vol 2(2):111–131, 2003.

[Neill03] C. J. Neill and P. A. Laplante. Requirements engineering: The state of the practice. IEEE Software, vol 20(6):40–45, 2003.

[OMG06] OMG. UML OCL Specification, v2.0. OMG Formal Document 2006-05-01, May 2006.

[OMG07] OMG. UML Superstructure Specification, v2.1.2. OMG Formal Document 2007-11-02, November 2007.

[Omondo] Omondo eclipseuml. http://www.omondo.de/.

[Rational] Rational Software Architect. http://www-306.ibm.com/software/awdtools/architect/swarchitect/.

[Reuys06] A. Reuys, S. Reis, E. Kamsties, and K. Pohl. The scented method for testing software product lines. In Käkölä, T.; Duenas, J.C. (Eds.): Software Product Lines - Research Issues in Engineering and Management, pages 479–520, Heidelberg, Springer, 2006.

[Rolland98] C. Rolland and C. B. Achour. Guiding the construction of textual use case specifications. In Data & Knowledge Engineering, volume 25 no. 1-2, pages 125–160, March 1998.

[Ryser00] J. Ryser and M. Glinz. SCENT: A Method Employing Scenarios to Systematically Derive TestCases for System Test. Technical report 2000.03, Institut für Informatik, University of Zurich, 2000.

[Spivey92] J. Spivey. The Z Notation: A Reference Manual. Prentice Hall, 1992.

[UCED] Use Case Editor (UCEd). http://sourceforge.net/projects/uced/.

[ViPER] ViPER project site. http://www.viper.sc.

[Whittle06] J. Whittle and P. K. Jayaraman. Generating Hierarchical State Machines from Use Case Charts. In RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), pages 16–25, Washington, DC, USA, IEEE Computer Society, 2006.

[Williams05] C. Williams, M. Kaplan, T. Klinger, and A. Paradkar. Toward Engineered, Useful Use Cases. In Journal of Object Technology, Special Issue: Use Case Modeling at UML-2004, volume 4, August 2005, pages 45–57, 2005.

About the authors

Veit Hoffmann is currently doing his doing his Ph. D. at the Research Group Software Construction of RWTH-Aachen. His mayor field of interest is in Requirements Engineering especially the conception and evaluation of textual use case descriptions. He can be reached at veit.hoffmann@swc.rwth-aachen.de.

Horst Lichter is professor for computer science at RWTH Aachen University where he heads the Research Group Software Construction. His group is mainly doing research in the area of model-based software development, quality assurance and software processes. He can be reached at lichter@swc.rwth-aachen.de.

Alexander Nyßen is a Ph.D. student at the Research Group Software Construction of the RWTH Aachen University. His primary research interest is in model-driven software engineering, with a special focus on related methods and tools. He can be reached at any@swc.rwth-aachen.de.

Andreas Walter has received his Diploma degree in Computer Science from RWTH Aachen University. Currently he is working as software engineer at sd&m AG, Troisdorf, Germany. There he is involved in information systems development projects, focusing on requirements engineering and software quality assurance. He can be reached at andreas.walter@sdm.de.