JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 8, No. 7, November–December 2009

A Meta-Model for Textual Use Case Description

Stéphane S. Somé, School of Information Technology and Engineering, University of Ottawa, Canada

A Use Case is a specification of interactions involving a system and external actors of that system. The capability for use case modeling has been integrated to the Unified Modeling Language (UML) since its inception. However, use cases are only defined at an abstract level, as the UML Specification does not discuss use case description in text form. In this paper, we propose an abstract syntax for textual use case description as a meta-model extension of the UML Specification. This meta-model is based on elements commonly found in use case templates. The meta-model also includes OCL constraints for ensuring consistency with the UML specification.

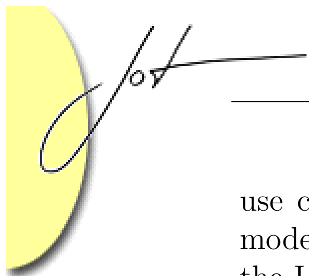
1 INTRODUCTION

The Unified Modeling Language (UML) defines a use case as "*the specification of a sequence of actions, including variants that a system (or a subsystem) can perform, interacting with actors of the system*" [12]. Use cases are used to drive the development process from the early stages of business modeling to acceptance testing [7]. The UML defines use cases at an abstract level by only providing an external view of use cases. The UML meta-model specifies the types of relations that a use case may have with other use cases or actors in the environment. However, the definition of how a use case concrete behavior (the use case sequence of actions) is specified is left open. The UML Specification suggests the concrete behavior corresponding to a use case, to be separately specified using various behavior description approaches such as interactions, activities, state machines, pre/post-conditions or natural language text.

The practical usage of use cases as advocated by software development methodologies such as the Unified Process [7] start with use cases description as natural language text. This form of behavior description is seen as better suited at the early stages of software development when business and user requirements are captured. One reason is that natural language is more accessible to stake-holders and therefore, allows straightforward validation. Other description formalisms are typically used at later development stages to refine and detail textual use cases.

Unlike formalisms such as interactions, activities and state machines, the UML does not formally specify a meta-model for a natural language notation of use cases. Different templates for use case description that can be considered as fulfilling this role exist [8] [2] [3] [6] [7] [1]. These templates provide guidelines for structuring

Cite this document as follows: Stéphane S. Somé: *A Meta-Model for Textual Use Case Description*, in Journal of Object Technology, vol. 8, no. 7, November–December 2009, pages 87–106,
http://www.jot.fm/issue/issues_2009_06/article3/



use case description. However, there is no formal connection to the UML meta-model that would ensure that the described use cases satisfy constraints specified in the UML Specification. For instance, a UML use case diagram shows relationships between use cases and the actors that participate in that use case. In order to remain consistent, a textual description formalism should therefore be able to enforce that actors referred within the use case description text must be among the actors related to the use case in the use case model. This type of consistency rule can be defined as part of a meta-model.

In this paper, we present a meta-model for textual use case description. The elements of the meta-model are based on an examination of use case templates such as [8], [2], [3], [6] and [7]. We also specify constraints using the Object Constraint Language (OCL) [11], aimed at aligning use case description with the UML Specification. The meta-model thus allows the definition and enforcement of consistency constraints defined in the UML Specification on textual use cases description. Other benefits provided by a meta-model include the potential for automated support for use case edition and generation of other behavior models such as activity diagram or statecharts, from use cases.

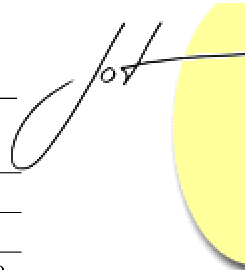
The remainder of the paper is organized as follow. We identify elements needed for use case description in the next section. This is based on a review of some commonly used use case description templates. In section 3, we introduce our meta-model along with OCL constraints for use case consistency. We also outline a concrete natural language syntax and informal semantics. Section 4 presents two implementations aimed at validating the meta-model. Some related works are discussed in section 5 and finally, section 6 concludes the paper.

2 ELEMENTS OF TEXTUAL USE CASE DESCRIPTION

Several templates have been proposed for textual use case description. Most of these are organization specific. Examples of published templates include [8], [2], [3], [6], [7] and [1]. Tables 1 and 2 describe the elements of two prevalent templates; the Rational Unified Process use case template[7] and Cockburn's use case template [2]. As can been seen from Tables 1 and 2, there are lots of variation regarding the elements of a use case description. In spite of the variations, two main parts can be distinguished in all reviewed templates: a *static part* and a *dynamic part*. The static part includes elements pertaining to the system's state (*preconditions* and *postconditions*) as well as other descriptive *traits* (e.g. *actors*, *description*, *priority*, ...). The dynamic part captures the use case behavior. It consists in a *trigger*, a *main sequence* of steps and none or several *alternatives* to steps.

The flow of execution of steps within a sequence of steps is governed by different types of control flow structures. The following are the most common control flow structures found in use case templates.

- Sequence: when steps un-conditionally follow each other. This type of flow is



Element	Description
Name	The name of the use case.
Brief Description	A brief description of the role and purpose of the use case.
Flow of Events	A textual description of what the system does in regard to the use case (not how specific problems are solved by the system). The description is understandable by the customer.
Special Requirements	A textual description that collects all requirements, such as non-functional requirements, on the use case, that are not considered in the use-case model, but that need to be taken care of during design or implementation.
Preconditions	A textual description that defines a constraint on the system when the use case may start.
Postconditions	A textual description that defines a constraint on the system when the use cases have terminated.
Extension points	A list of locations within the flow of events of the use case at which additional behavior can be inserted using the extend-relationship.

Table 1: Rational Unified Process use case description template.

implicitly determined by the ordering of steps.

- Alternative: when a step execution is conditional. Alternative flow structures are generally introduced as `if` like statements.

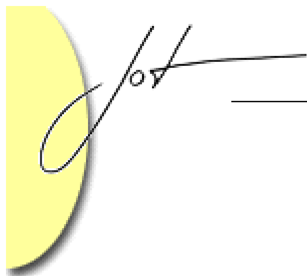
- Iteration: when a sequence of steps repeats based on a condition. Iterative structures are generally introduced similarity to `repeat`/`while` loops in programming languages.

- Concurrency: when different blocs of steps can execute in parallel.

The different types of steps include *actions* from actors or the system under consideration, and directives such as *use case inclusion* or *branching*.

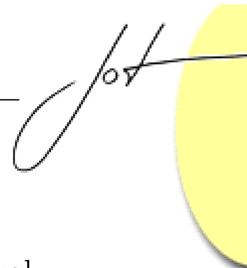
3 USE CASE DESCRIPTION META-MODEL

In this section, we first describe a meta-model for use case description. Then, we briefly sketch some informal semantics, introduce a concrete natural language-based syntax and present an example.



Element	Description
Number	The use case number
Name	Should be the goal as a short active verb phrase
Goal in Context	Longer statement of the goal, if needed
Scope	What system is being considered black-box under design
Level	One of: Summary, Primary task, Subfunction
Preconditions	What we expect is already the state of the world
Success End Condition	The state of the world upon successful completion
Failed End Condition	The state of the world if goal abandoned
Primary Actor	A role name for the primary actor, or description
Trigger	The action upon the system that starts the use case, may be time event
Main Success Scenario	The steps of the scenario from trigger to goal delivery, and any cleanup after <step #> <action description>
Extensions	Extensions, each referring to the step of the main scenario <step altered> <condition> : <action or sub.use case>
Sub-Variations	Sub-variations that will cause eventual bifurcation in the scenario <step or variation #> <list of sub-variations>
Priority	How critical to system / organization
Performance Target	Amount of time this use case should take
Frequency	How often it is expected to happen
Superordinate Use Case	Optional name of use case that includes this one
Subordinate Use Cases	Optional, depending on tools, links to sub.use cases
Channel to primary actor	e.g. interactive, static files, database
Secondary Actors	List of other systems needed to accomplish use case
Channel to Secondary Actors	e.g. interactive, static, file, database, timeout
Open Issues	Optional list of issues about this use cases awaiting decisions
Due Date	Date or release of deployment

Table 2: Cockburn's template for fully dressed use cases.



Meta-model description

Our approach consists in extending the UML meta-model with elements for textual use case description. Figure 1 shows the UML meta-model for use cases. Use cases

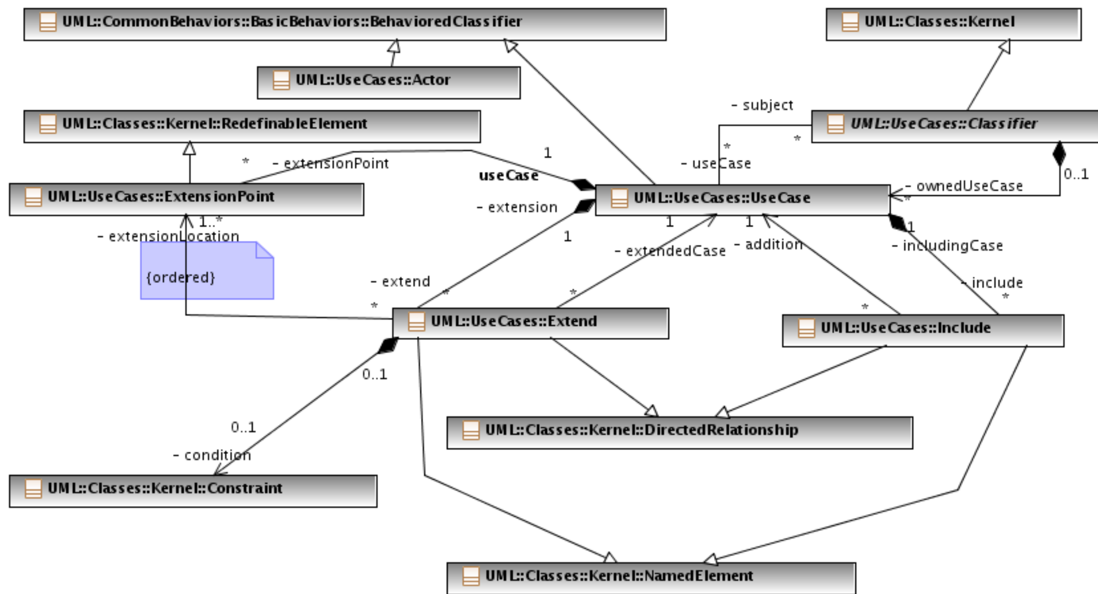


Figure 1: Use Case meta-model in the UML Specification

are specified at an abstract level along with actors and relationships. A use case is a sub-class of `BehavioredClassifier`. As such it may *own* behaviors [12]-p432.

Class `UseCaseDescription` shown in Figure 2, captures a use case textual description. We define `UseCaseDescription` as a specialization of meta-class `Behavior`. Therefore, instances of `UseCaseDescription` may be among the behaviors own by an instance of `UseCase`. We allow a use case to be associated to more than one use

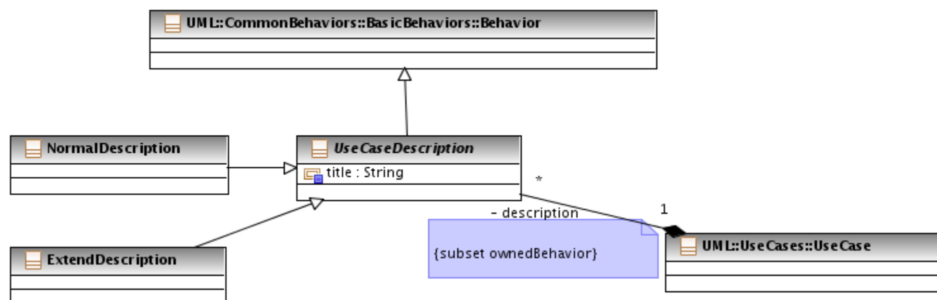
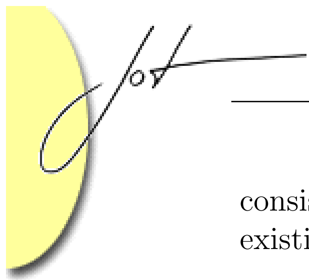


Figure 2: Variants of use case description

case description. In practice several textual descriptions may be maintained for a same use case showing for instance different levels of detail. We acknowledge that



consistency among the different descriptions is an issue. However, this is already an existing problem given that a `BehavioredClassifier` may owe several behaviors.

We distinguish two subclasses to `UseCaseDescription`: `NormalDescription` and `ExtendDescription`. A `NormalDescription` specifies a "traditional" use case, while an `ExtendDescription` is used for an extension use case. The distinction is needed as the UML Specification mentions that an "*extending use case defines a set of modular behavior increments that augment an execution of the extended use case under specific conditions*" [12]-p589. In order to be consistent with this statement, an `ExtendDescription` shall be able to be associated with a set of independent behavior definitions whereas a `NormalDescription` specifies a single behavior chunk.

The distinction between normal and extend description introduces the following constraints.

OCL1 A textual description of an included use case (a use case target of an <<*include*>> relation) must be an instance of `NormalDescription`.

This corresponds to the following OCL statement.

```
context UML::UseCases::Include inv:
 self.addition.description->forall(d | d.oclIsTypeOf(NormalDescription))
```

OCL2 A textual description of an extending use case (a use case source of an <<*extend*>> relation) must be an instance of `ExtendDescription`.

```
context UML::UseCases::UseCase inv:
  self.extend->size() > 0 implies
    self.description->forall(d | d.oclIsTypeOf(ExtendDescription))
```

Figure 3 shows a description of meta-class `NormalDescription`. In light of our review of use case templates summarized in section 2, a `NormalDescription` includes a *static part* and a *dynamic part*. The static part includes descriptive traits. The actors involved in a use case are denoted using traits *primary actor* and *participants*. Since actors are specified at the use case model level, the actors referred to in the *primary actor* and *participants* traits must correspond to actors related to the use case.

OCL3 A use case primary actor must be among the actors related to that use case.

```
context UseCase inv:
 self.description.oclIsTypeOf(NormalDescription) implies
  self.description.oclAsType(NormalDescription).primaryActor.actor.
      ownedAttribute->exists(a | a.opposite.class = self)
```

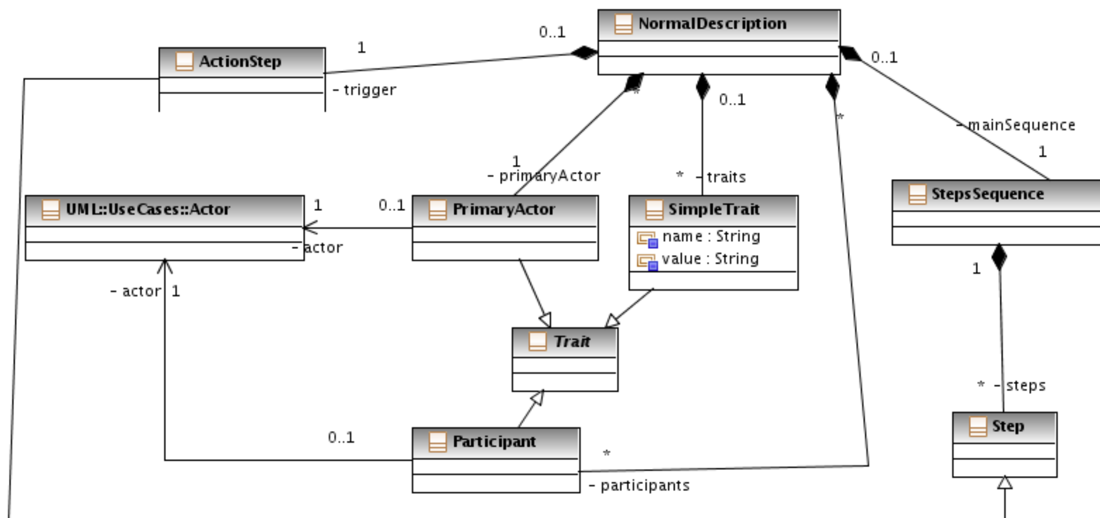


Figure 3: Description of meta-class NormalDescription

OCL4 All use cases secondary actors must be among the actors related to that use case.

```
context UseCase inv:
 self.description.oclIsTypeOf(NormalDescription) implies
  self.description.oclAsType(NormalDescription).particitants->
   forall(p | p.actor.ownedAttribute->
       exists(a | a.opposite.class = self))
```

Meta-class `UML::CommonBehaviors::BasicBehaviors::Behavior` of which `NormalDescription` is a subclass, has two associations named *precondition* and *postcondition* to members of type `UML::Classes::Kernel::Constraint` [12]-p430. These two associations are used for normal use cases description elements pertaining to the system's state. The set of preconditions describes the state in which the system needs to be before the use case can be executed, while the set of postconditions describes the state of the system at the successful completion of a use case. Preconditions and postconditions must be specified for a normal use case. On the other hand, since an extension use case defines behavior chunks that are typically independent and not necessarily meaningful on their own, preconditions and postconditions should not be specified for an extend description.

OCL5 The preconditions and postconditions of a normal description must not be empty.

```
context UseCase inv:
```

```
self.description.oclIsTypeOf(NormalDescription)
implies
   self.precondition->size() > 0 and self.postcondition->size() > 0
```

OCL6 The preconditions and postconditions of an extend description must be empty.

```
context UseCase inv:
 self.description.oclIsTypeOf(ExtendDescription)
 implies
    self.precondition->size() = 0 and self.postcondition->size() = 0
```

Because of the large variability of traits used in a use case description, the remaining description traits of a use case are captured as instances of `SimpleTrait`. This allows flexibility in customizing a use case description with particular traits, at the expense of the possibility for a more formal treatment of these traits. For instance, elements such as *Goal*, *Scope*, or *Level* from Cockburn's template in Table 2, would correspond to simple traits with appropriate values for attributes *name* and *value*.

The dynamic part of a normal use case description includes a *trigger* (instance of `ActionStep`) and a `StepsSequence` that represents the *main sequence of steps*. We distinguish different types of steps as shown in Figure 4. Each step may be con-

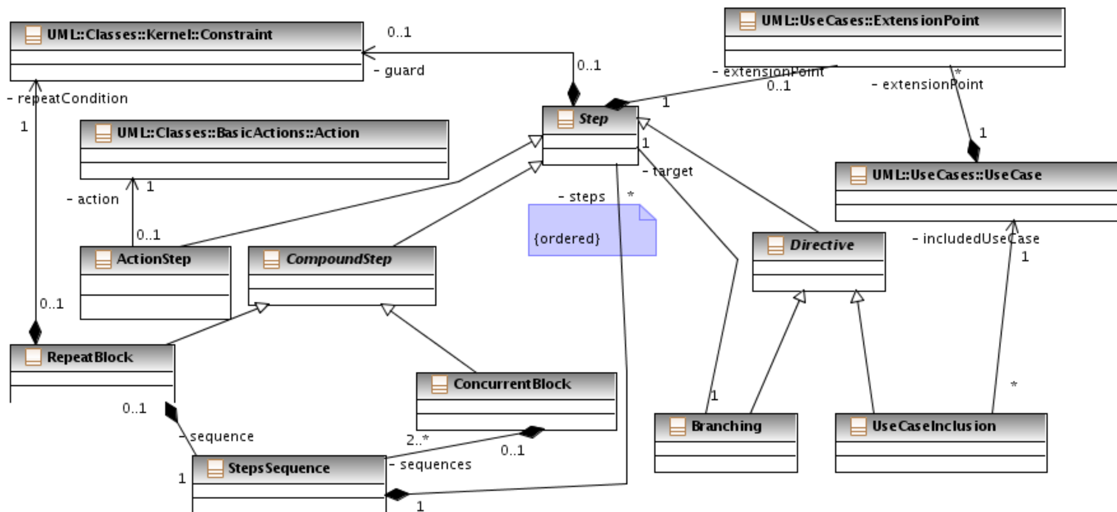


Figure 4: Use Case steps

strained by a *guard* and may be the location of an *extension point*. An `ActionStep` is a basic use case step corresponding to an *action* (an instance of `UML::Classes::-BasicActions::Action`). An action is associated to a *context*; a classifier that owes the behavior of which the action is a part [12]-p237. In addition to action steps, a

use case may specify *directives* and *compound steps*. Directives include *branching* (`Branching`) and *use case inclusion* (`UseCaseInclusion`). A branching is used to redirect a sequence of execution flow, while a use case inclusion expresses the realization of an <<include>> relation within a use case description. *Compound steps* are grouping of steps according to specific control flow structures. In accordance with our review of use case templates, we distinguish *repeat blocks* (`RepeatBlock`) for iterative steps controlled by a constraint (*repeatCondition*), and *concurrent blocks* (`ConcurrentBlock`) for blocs of steps which execute in parallel.

Alternatives may be attached to action steps as shown in Figure 5. An alternative

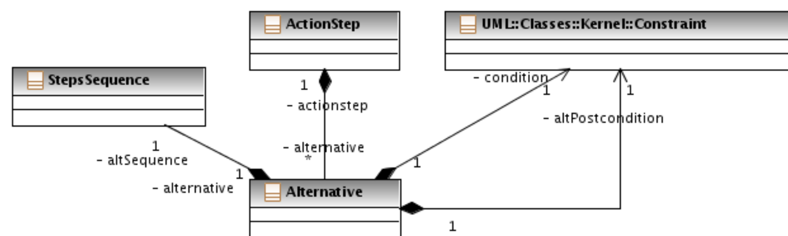


Figure 5: Action steps

specifies a *variation* in the course of execution of a use case. Alternatives typically correspond to exceptional/error situations or other ways to achieve a use case goal. An alternative is based on a given *condition* that is set according to the outcome of the action steps to which it is attached.

Figure 6 shows the meta-model for extension use case description (`ExtendDescription`). An extension use case specifies a set of *fragments* each consisting of

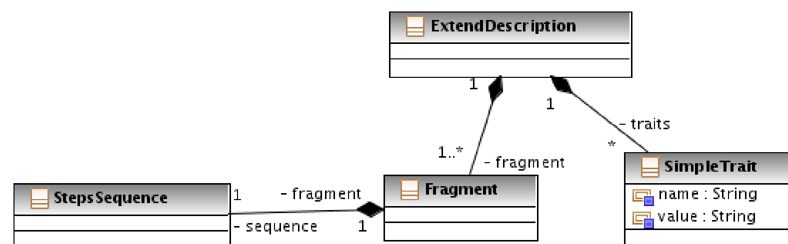


Figure 6: Meta-model for Extend Description

a steps sequence. Additionally, different descriptive traits may be attached to the description.

Following are other constraints related to use case descriptions.

OCL7 A normal use case trigger must belong to the use case primary actor.

```
context UseCase inv:
 self.description.oclIsTypeOf(NormalDescription) implies
   self.description.oclAsType(NormalDescription).trigger.action =
    self.description.oclAsType(NormalDescription).
        primaryActor.actor
```

This constraint is motivated by the fact that according to the UML specification, a (normal) use case specifies a unit of functionality which is initiated by an actor [12]-p594 (the primary actor).

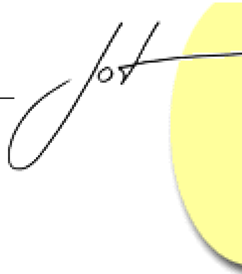
OCL8 The extension points specified within a use case description must correspond to extension points defined at the use case model level.

```
context UseCase inv:
 self.allSteps()->forall(s |
  s.oclIsTypeOf(ActionStep) and not s.extensionPoint.oclIsUndefined()
   implies self.extensionPoint->includes(s.extensionPoint))
```

Query allSteps() returns all the steps of a use case by flattening steps sequences.

```
UseCaseDescription::allSteps():Set(Step)
 if self.description.oclIsTypeOf(NormalDescription)
  then
    self.oclAsType(NormalDescription).mainSequence.allSteps()
  else
    if self.description.oclIsTypeOf(ExtendDescription)
     then
       self.oclAsType(ExtendDescription).parts->collect(p
                                  | p.sequence.allSteps())->flatten()
    endif
 endif


StepsSequence::allSteps():Set(Step)
 if steps->isEmpty()
  then Set{}
  else
    steps->collect(s |
      if s.oclIsTypeOf(ActionStep)
        then
        s.oclAsType(ActionStep).altSequence.allSteps()->
                                                    including(s)
        else
          if s.oclIsTypeOf(RepeatBlock)
          then s.oclAsType(RepeatBlock).sequence.
```



```
                                                allSteps()->including(s)
                    else
                      if s.oclIsTypeOf(ConcurrentBlock)
                      then
                      Set{}->including(s)->union(s.oclAsType(ConcurrentBlock).
                                  sequences->collect(s | s.allSteps()))
                      else
                        Set{}->including(s)
                      endif
                    endif
                  endif
                endif)->flatten()
            endif
```

OCL9 An *included* use case referred by a use case inclusion directive within an *including* use case description, and that including use case must be related by an <<*include*>> relation at the use case model level.

```
context UseCase inv:
  self.description.allSteps()->forall(s |
    s.oclIsTypeOf(UseCaseInclusion) implies
      self.include->includes(s.oclAsType(
                        UseCaseInclusion).includedUseCase))
```

OCL10 The number of fragments of an extension use case description should be at least equal to the number of extension points referred in each <<*extend*>> relation from that extension use case.

```
context UseCase inv:
  self.description.oclIsTypeOf(ExtendDescription) implies
    self.extend->forall(e | e.extensionLocation->size() <=
      self.description.oclAsType(ExtendDescription).fragment->size())
```

This constraint is needed to remain consistent with the UML Specification. An <<*extend*>> relation refers to "*an ordered list of extension points belonging to the extended use case, specifying where the respective behavioral fragments of the extending use case are to be inserted. The first fragment in the extending use case is associated with the first extension point in the list, the second fragment with the second point, and so on*" [12]-p590. The number of fragments defined in the extending use case must therefore be at least equal to the number of extension points referred to by the relation. Notice that the reverse is not required as "extra" fragments would just be ignored.

Informal Semantics

According to the UML specification, "*a UseCase is a kind of behaviored classifier that represents a declaration of an offered behavior*" [12]-p594. A textual representation of a normal use case captures that behavior through a *trigger* followed by a *mainSequence* of steps. The `Actions` associated to instances of `ActionSteps` are the basic elements of a use case behavior. The ordering of steps within steps sequences determines actions sequencing. The basic flow scheme is that if step *i* is followed by step *i+1*, then an action corresponding to step *i+1* would follow the successful completion of an action corresponding to step *i*. *Compound steps* and *directives* introduce some alteration to this basic scheme.

- The last step of a `RepeatBlock` steps sequence is followed by the first step when the *repeatCondition* holds, and is followed by the step right after the `RepeatBlock` if the *repeatCondition* does not hold.
- The behaviors captured by the different step sequences of a `ConcurrentBlock` are *interleaved*. Actions from a same step sequence are sequentially ordered as usual, but actions from different sequences are not strictly ordered.
- The step following a *branching* directive is the target of that directive. As a consequence, all steps after a *branching* directive in a same steps sequence are unreachable.

The UML semantics for use case inclusion and extension are preserved as such.

An example of concrete syntax

We present elements of a concrete syntax corresponding to the use case description meta-model. The interested reader is referred to [15] for a more complete description of this concrete syntax. We should stress that the notation discuss here is an example. One of the motivations of the meta-model being to allow development of customized concrete syntaxes, different conforming use case description notations are possible.

The presentation is illustrated with a use case model for a Broker System. A use case diagram for this system is shown in Figure 7. The goal of the Broker System is to allow customers to find the best supplier for a given order. A customer fills up an online order form and after submission, the system broadcasts it to suppliers. Each supplier after examining the order may decide to decline or submit a bid. Submitted bids are sent back to the broker to be shown to the customer, who eventually asks the system to proceed with a bid. The elements of the use case model in Figure 7 are instances of the UML meta-model for use cases shown in Figure 1. For instance, the system under design *Broker System*, is a `Classifier`. All use cases are instances of meta-class `UseCase` owned by *Broker System*. The model also

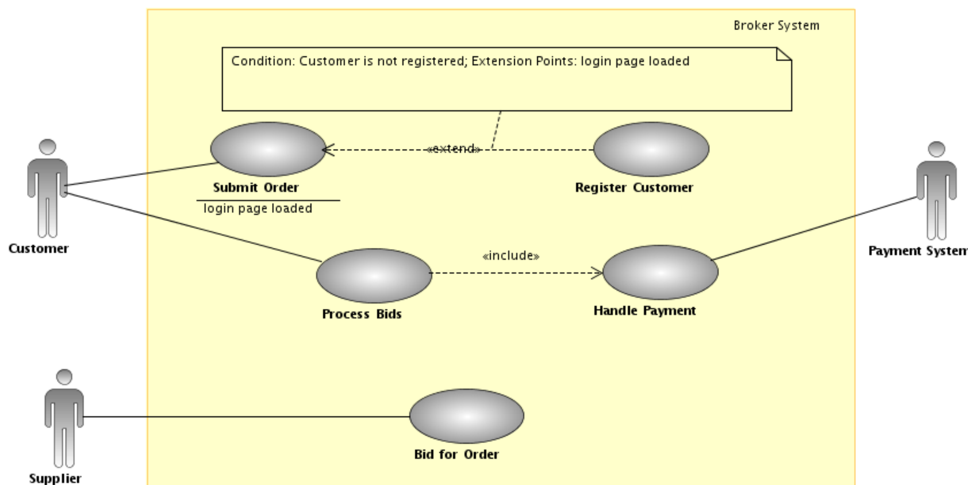


Figure 7: Broker System use case model

comprises an <<*include*>> relation from use case *Process Bids* to *Handle Payment* and an <<*extend*>> relation from *Register Customer* to *Submit Order*. The later relation depends on condition *Customer is not registered* and refers to extension point *login page loaded* that belongs to use case *Submit Order*.

Figures 8 and 9 show the textual description of two of the *Broker System* use cases. The description is based on a concrete notation defined using a restricted form of natural language. Use case *Submit Order* is a normal use case which description is an instance of meta-class `NormalDescription` (Figure 3). The different elements of a use case are identified by a corresponding section heading. Section titled *Description* corresponds to a `SimpleTrait`. The value of that trait is an unconstrained text.

We use *condition sentences* as concrete representation of constraints such as preconditions, postconditions or guards. A condition sentence describes a *situation* prevailing within a system and environment. It may be a *simple condition sentence*, a negation of a condition sentence, or a combination of condition sentences using conjunctions/disjunctions. A simple condition sentence adheres to the format

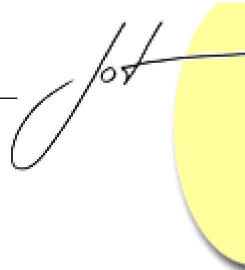
"name of entity" "verb" "possible value of entity"

with "verb" a conjugated form of a limited number of verbs including to be and to have. A *domain model* that enumerates all the entities in the application and their possible values is needed for parsing. A substantial part of this model is obtained by pre-processing use cases [10]. For instance the preconditions of use case *Submit Order* "The Broker System is online and the Broker System welcome page is being displayed" is a condition sentence consisting in a conjunction of two simple conditions. The first simple condition involves entity "Broker System", verb "is" (a conjugated from of "to be") and possible value "online".

The main sequence of a normal use case is specified in section titled STEPS. The optional *guard* of a step is introduced with keywords IF ... THEN as in step *4* of use

```
Title: Submit Order
Description: This use case describes a process through which
      a Customer using the Broker System, create an Order consisting
      in a set of Items, and broadcast it to potential Bidders.
Primary Actor: Customer
Preconditions: The Broker System is online and the Broker System
  welcome page is being displayed
Postconditions: An Order has been broadcasted
Trigger: The Customer loads the login page
STEPS
  1. The Broker System asks for the Customer's login information
  2. The Customer enters her login information
  3. The Broker System checks the provided login information
  4. IF The Customer login information is accurate
    THEN The Broker System displays an order page
  5. The Customer creates a new Order
  6. Repeat while the Customer has more items to add
    to the Order
    6.1. The Customer selects an item
    6.2. The Broker System adds the selected item to the order
  7. The Customer submits the Order
  8. The Broker System broadcast the Order to the Suppliers
ALTERNATIVES
  3a. The Customer login information is not accurate
     3a1. GOTO Step 1.
  7a. The Order is empty
     7a1. The Broker System displays an error page
EXTENSION POINTS
  STEP 1. login page loaded
```

Figure 8: Description of use case “Submit order” in the Online Broker System.



```
Title: Register Customer
Description: This use case describes additional behavior
      triggered when registration is needed for a new Customer.
FRAGMENT 1.
   1.1. Customer selects registration operation
   1.2. Broker System asks for Customer name, date of birth and address
   1.3. Customer enters registration information
   1.4. Broker System validates Customer information
   1.5. Broker System generate login information for Customer
   ALTERNATIVES
      1.4.a. Customer registration information is not valid
      1.4.a.1. Broker System displays registration failure page
```

Figure 9: Extension use case "Register Customer".

case *Submit Order*.

The trigger and all steps except step *6* in use case *Submit Order* are action steps instances of meta-class `ActionStep`. An action step denotes the execution of an operation triggered by an actor in the environment of the system, or the execution of an operation initiated by system in reaction to an actor's action. For instance, the trigger of use case *Submit Order* is an action step denoting the execution of an operation triggered by actor Customer (the context of the action). On the other hand, step *1* corresponds to an action executed in the context of the Broker System in reaction to the use case trigger. Our concrete syntax [15] for action steps is based on the declaration of operations in the domain model according to the format "action_verb [action_object]"[1]. Where the action_verb is a verb in infinitive and the action_object refers to an entity. As an example, "load login page" is an operation name where the action verb is "load" and the action object is "login page". Given this naming convention, an action step has the following form:

"name of concept" "action_specification" ["preposition" "action_participant"]

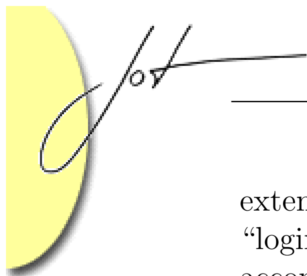
The "action_specification" has the form

"conjugated_action_verb" ["action_object"]

The "conjugated_action_verb" is the "action_verb" used in the concept operation declaration in the present tense.

The alternatives to actions steps are detailed in the section titled ALTERNATIVES. In use case *Submit Order*, only steps *3* and *7* have associated alternatives. Each of the alternatives includes a condition (e.g. "The Customer login information is not accurate") and a sequence of steps. Action steps may also be associated to

[1]Elements between "[]" are optional.



extension points. For instance, step *1* is associated to an extension point labeled "login page loaded". The label is used for matching in <<*extend*>> relations in accordance with constraint OCL8.

Step *6* of use case *Submit Order* is a *repeat block* instance of meta-class `RepeatBlock`. We introduce repeat blocks with keywords Repeat while followed by the *repeat condition*. The iterated step sequence consists in steps *6.1* and *6.2*. A *concurrent block* is introduced with keywords In Parallel as follow.

```
X. In Parallel
  X.1.1. ...
  X.1.2. ...
  ...
 AND
  X.2.1. ...
  X.2.2. ...
  ...
 AND
  ...
```

Directives are similarly introduced using keywords. For instance, step *3a1.* is a *branching directive* representing an instance of meta-class `Branching` specified with keyword GOTO. The target of that branching directive is step *1*. A *use case inclusion directive* is specified using keyword include in accordance to format "include *use_case_name*".

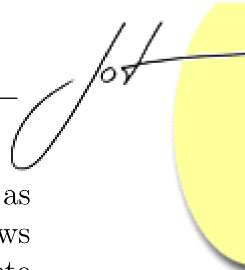
Figure 9 shows an extension use case. This use case includes a simple trait titled *Description* and a single fragment (instance of meta-class `Fragment`). The fragment specifies a step sequence intended to extend the behavior of extended use cases according to <<*extend*>> relations.

4 IMPLEMENTATION

The meta-model described in this paper has been used as basis for two use case modeling tools. The first tool is an Eclipse plugin developed using the Eclipse Modeling Framework (EMF)[2] and the Eclipse Model Development Tools (MDT)[3]. EMF automates the generation of editors from models while the MDT includes a reusable EMF-based implementation of the UML meta-model as well as an implementation of the OCL for EMF models. The resulting tool is a very basic use case editor with limited usability. However, this implementation allowed us to connect our meta-model to the UML meta-model and validate the OCL constraints.

The Use Case Editor (UCEd) is a more elaborate use case modeling tool based

[2] http://www.eclipse.org/modeling/emf/
[3] http://www.eclipse.org/modeling/mdt/?project=uml2



on the meta-model[4]. UCEd provides use case modeling and editing facilities as well UML StateChart [12] generation and use case simulation. Figure 10 shows a view of UCEd use case editing tool. UCEd accepts use cases in the concrete

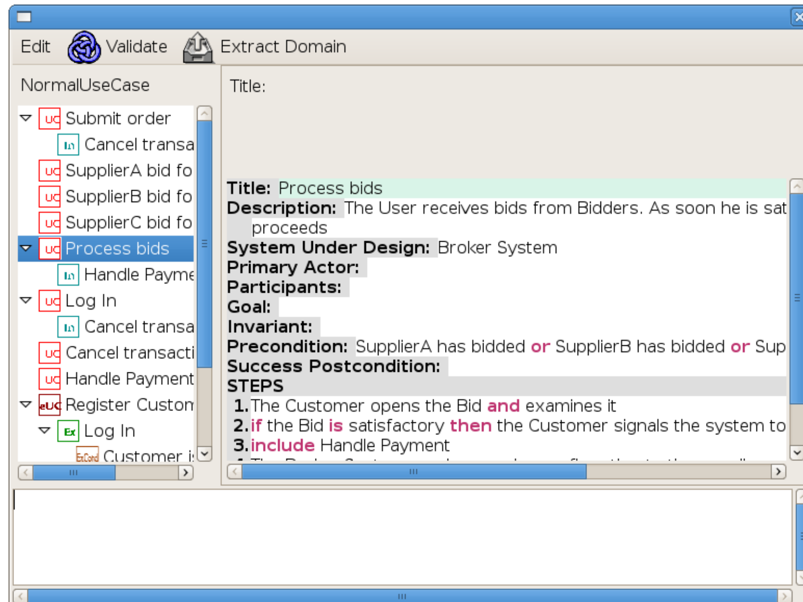


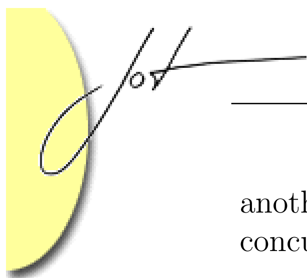
Figure 10: Use Case edition in UCEd

syntax outlined in this paper, parses them and generates a StateChart equivalent of the modeled behavior. The use cases are parsed using a domain model and the generated StateChart can be used as prototypes to animate and validate the use cases.

5 RELATED WORK

Meta-models for textual use case description have been proposed in various works. In [5], an approach for the generation of UML Activity diagrams [12] from textual use cases is presented. The generation is formulated as a set of transformation rules defined at use cases and activity diagrams meta-model levels. Attributes of a use case (referred as an instance of a meta-class called `FunctionalRequirement`) include elements such as *preconditions*, *postconditions*, *description* in addition to a *main sequence* of actors' actions. Each action may be associated to *exceptional steps*. A significant distinction between our meta-model and the one in [5], lies in the fact that our meta-model is formally defined as an extension to the UML specification. We provide a connection between use case description and UML use case models by introducing a set of constraints that serve to ensure consistency between use case descriptions and use case models. The degree of expressiveness offered can be seen as

[4] http://www.site.uottawa.ca/ ssome/Use_Case_Editor_UCEd.html



another difference as our meta-model introduces control flow structures (iteration, concurrency) and allows the definition of variable custom traits.

A use case meta-model is introduced in [4] as part of an XML-based approach for requirements verification. The meta-model distinguishes the following as attributes of a use case: *triggering event*, *precondition*, *postcondition* and *frequency*. A use case also includes a sequence of *steps*. Each step refers to an *action*, a set of *exceptions* and an optional *condition*. The meta-model distinguishes *actor's actions*, the *system's actions* and *use case actions* such as use case inclusion. The differences between [4] and our work are similar to those with [5]; lack of connection to the UML Specification, limitation to expressiveness.

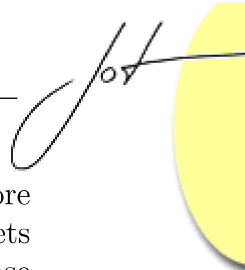
An approach for refactoring use case models based on a use case meta-model is discussed in [14]. The meta-model distinguishes three levels: an *environmental level* where the concept of *use cases* is defined and related to external elements such as *actors*, *goals* and *users*; a *structure level* that defines the internal structure of use cases in terms of *preconditions*, *postconditions*, *scenarios* and *episodes*; and an *event level* where the different types of events (*stimuli*, *responses*, *internal actions*) making up an episode are distinguished. It is not clear from [14] if the proposed meta-model is intended for textual use cases. Beside that, the meta-model in [14] is not related to the UML Specification and is limited in term of expressiveness.

In [18], a use case meta-model consistent with version 1.3 of the UML Specification is presented. This work does not specifically deals with textual use cases, but rather provides a more formal treatment of use case meta-model than the one in the UML Specification, with the improvement of the testability of use case models as objective. The Object Constraint Language is used to specify well-formedness rules on the meta-model.

Another work on use case meta-modeling is discussed in [9]. This work presents a requirements description meta-model that integrates the UML activity graph meta-model and the UML use case meta-model. Activity graphs are used for use cases description. The meta-model thus formally connects use case descriptions to use case models, which constitutes a similarity between our work and [9].

6 CONCLUSIONS

According to [17], the lack of well defined semantics is one of the limitations to a wider adoption of use cases in industry. In this paper, we presented a meta-model for textual use case description that could serve as a definition of static semantics. Our meta-model is defined as an extension to the UML Specification. Although textual use case description is not discussed in the current version of the UML, the ubiquity of that formalism for use case description contributes to the significance of our work. The constraints associated to the meta-model would allow ensuring the consistency between textual use cases and use case models.



We only provided an informal overview of dynamic semantics here. A more formal treatment of the dynamic semantics of textual use cases based on Petri nets [13], is discussed in [16]. We also outlined a concrete natural language-based use case description syntax built on top of the meta-model. In our future work, we intend to further extend our notation in order to improve use case authoring flexibility.

REFERENCES

[1] Kurt Bittner and Ian Spence. *Use Case Modeling*. Addison-Wesley, 2003.

[2] A. Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2001.

[3] Derek Coleman. A Use Case Template: Draft for discussion. Fusion Newletter, http://www.bredemeyer.com/pdf_files/use_case.pdf, April 1998.

[4] Amador Durán, Antonio Ruiz-Cortés, Rafael Corchuelo, and Miguel Toro. Supporting requirements verification using xslt. *IEEE International Conference on Requirements Engineering*, 0:165, 2002.

[5] Javier J. Gutiérrez, Clémentine Nebut, María J. Escalona, Manuel Mejías, and Isabel M. Ramos. Visualization of use cases through automatically generated activity diagrams. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 83–96, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] ITU-T. Management interface specification methodology. International Telecommunication Union, Recommendation M.3020, July 2007.

[7] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1998.

[8] Ruth Malan and Dana Bredemeyer. Functional Requirements and Use Cases. http://www.bredemeyer.com/pdf_files/functreq.pdf, June 1999.

[9] Takako Nakatani, Tetsuya Urai, Sou Ohmura, and Tetsuo Tamai. A requirements description metamodel for use cases. In *APSEC '01: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, page 251, Washington, DC, USA, 2001. IEEE Computer Society.

[10] S. Nayanamana and S. Somé. Generating a Domain Model from a Use Case Model. In *14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2005)*, july 2005.

[11] OMG. Object Constraint Language Version 2.0. http://www.omg.org, May 2006.

[12] OMG. UML Superstructure Specification, v2.1.2. http://www.omg.org, November 2007.

[13] Carl A. Petri. *Communication with Automata*. PhD thesis, Technische Universitat Darmstadt, 1962.

[14] Kexing Rui and Greg Butler. Refactoring Use Case Models: The Metamodel. In *Proceedings of the 25 th Australasian Computer Society Conference (ACSC 2003*, pages 301–308, 2003.

[15] S. Somé. Supporting Use Cases based Requirements Engineering. *Information and Software Technology*, 48(1):43–58, 2006.

[16] S. Somé. Petri Nets Based Formalization of Textual Use Cases. Technical Report TR-2007-11, SITE, University of Ottawa, 2007. http://www.site.uottawa.ca/eng/school/publications/techrep/2007/TR-2007-11.pdf.

[17] Clay Williams, Matthew Kaplan, Tim Klinger, and Amit Paradkar. Toward Engineered, Useful Use Cases. *Journal of Object Technology*, 4(6):45–57, August 2005.

[18] Clay E. Williams. Towards a test-ready meta-model for use cases. In Andy Evans, Robert B. France, Ana M. D. Moreira, and Bernhard Rumpe, editors, *pUML*, volume 7 of *LNI*, pages 270–287. GI, 2001.

ABOUT THE AUTHORS

Stéphane S. Somé is an assistant professor at the School of Information Technology and Engineering (SITE), University of Ottawa, Canada. Dr. Somé obtained his Ph.D. from the University of Montréal in Canada. He currently teaches various courses related to software engineering. His research interests include requirements modeling, software validation, Web engineering and program comprehension. He can be reached at ssome@site.uottawa.ca.