A Configurable UML Based Use Case Modeling Metamodel

L'uboš Zelinka and Valentino Vranić
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technology
Slovak University of Technology,
Ilkovičova 3, 84216 Bratislava 4, Slovakia
zelinka04@student.fiit.stuba.sk, vranic@fiit.stuba.sk

Abstract

There is a variety of approaches to use case modeling, especially regarding their textual description as their true form. Under certain circumstances, the use of each one of these approaches may be justified. A consistent application of a particular approach requires the existence of its metamodel. It appears that use case modeling notations are close enough to each other to allow for constructing a common, configurable use case modeling metamodel. Such a metamodel is proposed in this paper. It is based on the UML metamodel elements relevant to use case adapted and extended to cover different use case modeling notations with a special attention given to the elements of use case description. The configuration options of the proposed use case modeling metamodel are identified and the actual values for Jacobson's and Cockburn's notation presented and discussed. The metamodel is evaluated by a configurable use case modeling tool prototype.

1. Introduction

Use cases are widely used, but often degraded to be merely UML diagrams with only a few words of description. Being UML standardized gives a false impression of uniformness of this technique. Indeed, speaking of use case diagrams, we may note that although there are other graphical notations ranging from those close to the UML use case notation [5, 8] to some quite different notations [4] in practice, they are truly a realm of UML. However, UML does not prescribe any notation for the use case description, which is their main form. There is a tremendous number of varieties in the use case textual description.

It is possible to discuss advantages of one approach to use case modeling over the other ones, but that would not bring any real value to modelers who have to use them as decisions that lead to adopting a particular approach might be beyond their control. They are interested in having their use case modeling as such—as well as consistency of the resulting models—supported by a tool. However, the tool support of a particular notation cannot possibly exist without making it clear what is, and what is not a part of the notation.

A model of the notation—i.e., a metamodel—is what is actually needed. It appears that use case modeling notations are close enough to each other to allow for constructing a common, configurable use case modeling metamodel, which is the topic of this paper.

The paper is further organized as follows. Section 2 analyzes differences in understanding and application of the use case notation elements focusing mainly on Jacobson's and Cockburn's notation. Based on this analysis, Sect. 3 introduces a use case modeling metamodel. Section 4 presents options for the configuration of this metamodel and their values for Jacobson's and Cockburn's notation. Section 6 discusses related work. Section 7 brings conclusions and directions of further work.

2. Diversity in use case modeling

A use case describes a coherent functionality that provides some result of value to a user. As the term says, it is a case of a system use [1]. There are many different ways of describing use cases, but all of them have their roots in Jacobson's or Cockburn's notation. This section briefly explains the most prominent elements of use case description as such, and than points out the differences between Jacobson's and Cockburn's notation. The way the technique is used in practice is to a large extent influenced by the capabilities of available tools, so an overview of tool support is presented, too.

2.1. Use case description

A semiformal use case description is simply a natural language text structured using a text template which divides the text into logical parts. Even though there is no broadly accepted standard use case template, the existing templates are quite similar.

Name and *brief description* provide the reader with basic information about the use case. The use case name—sometimes called *title*—uniquely identifies the use case in the use case model (or at least in its namespace if the model is partitioned). Use case names may be accompanied by identification numbers used to refer to them.

Actors are roles adopted by external entities that interact with the system directly [1]. Typically, actors are user roles, but systems, subsystems, or even time can all perform as actors. Each actor can participate in many use cases and each use case can embrace several actors. It is often distinguished between primary and secondary actors. Primary actors participate in a use case to satisfy their goals, while secondary actors help the system satisfy goals of primary actors.

Preconditions are a set of constraints that should be fulfilled before the use case starts. *Postconditions* are a set of constraints that would be fulfilled after the use case finishes if preconditions have been satisfied before it started. This is actually the design by contract [12], but we may encounter different, less restrictive understanding of preconditions and postcondition, putting them to a merely informative position [20], or a fully restricted view, where the very use case activation is presumed by fulfilling its precondition [1].

Flows of events—or simply just *flows* (known also as *scenarios*)— represent every possible outcome of an attempt to accomplish a use case goal [15]. A flow is a sequence of interactions between an actor and a system. The interactions start from the triggering action and continue until the goal is delivered or abandoned [10]. In the use case description the interactions are represented by steps. Flows are sometimes represented as prose, but usually they are represented as sequences of steps, or—more precisely—partial orderings of steps [5], as some steps simply don't fit into any ordering (such as "at any point, a user can cancel the activity"). Very often, it is distinguished between *main* (or *basic*) and *alternative* flows. A main flow describes the normal sequence of steps in the execution of a use case [2]. Usually, it represents the interaction between the actor and the system under ideal conditions without alternatives and exceptions. Alternative flows cover behavior that is of optional, exceptional, or truly alternate character in relation to another flow [2]. They are dependent on some condition occurring at an explicit point in another flow. Additionally, another category of flows can be distinguished: subflows, which are used to separate repetitive interaction from other flows [10].

Use case relationships are a part of the use case description even though they are not explicitly present in most of the use case templates. UML offers two standard relationships between use cases called *include* and *extend*. The include relationship defines that a use case contains the behavior defined in another use case [14]. The purpose of this relationship is to reuse existing behavior or extract identical behavior. The behavior of the included use case is simply inserted into the behavior described in the including use case. It is similar to a function call in a programming language, but the include relationship should not be used for a functional decomposition. The extend relationship is a relationship directed from the extending use case towards the use case being extended that specifies how and when the behavior defined in the extending use case can be inserted into the behavior defined in the use case being extended [14]. It is typically used to add optional or exceptional behavior without making changes to the behavior described in extended use case, which is similar to alternative flows.

The extend relationship is used in combination with *extension points*, which are named places in the flow of events where additional behavior can be inserted or attached [2]. Every flow of events can have multiple extension points. A common way to define an extension point in a use case description is inserting its name into the flow of events between two steps. It's a pretty straightforward way, but every extension point refers only to a single place in the flow of events, which can be limiting. Since steps in flows of events are usually numbered, it is possible to define the extension point by a step number. In addition, it is possible to create extension points that occur over multiple steps between two step numbers.

2.2. Jacobson's notation

Jacobson's and Cockburn's use case modeling notation are well established and distinguished. Consider the example in Fig. 1 and 2. We can see that Jacobson's notation allows multiple main flows. The extension point is defined by a step number in a specific flow which suggest the possibility of using extension points over multiple steps. Beside alternative flows and subflows, Jacobson employs a special flow denoted as *extension flow*, but it can be seen just as an alternative flow defined in another use case, as confirmed in Jacobson's own writing [10].

2.3. Cockburn's notation

Cockburn is a strong proponent of a purely textual representation of use cases. In the example of extend relationship in Fig. 3, the *Check spelling* use case extends the *Edit a document* use case implicitly by its main flow. There is no ex-

Use Case: Reserve Room
Basic Flows:
B1. Reserve Room
The use case begins when a customer wants to reserve a room.

1. The customer selects to reserve a room.
2. The system displays the types of rooms the hotel has and their rates.
3. The customer **Check Room Cost**.
4. The customer makes the reservation for the chosen room.
5. The system deducts from the database the number of rooms of the specified type available for reservation.
6. The system creates a new reservation with the given details.
7. The system displays the reservation confirmation number and check-in instructions.
8. The use case terminates.

Alternate Flows:
A1. Duplicate Submission If in step 5 of the basic flow there is an identical reservation in the system (same name, e-mail, and start and end dates), the system displays the existing reservation and asks the customer if he wants to proceed with the new reservation.

1. If the customer wants to continue, the system proceeds with the reservation, and the use case resumes.
2. If the customer indicates that the new reservation is a duplicate, the use case terminates.

Subflows:
S1. Check Room Cost

1. The customer selects his desired room type and indicates his period of stay.
2. The system computes the cost for the specified period.

Extension Points:
E1. Update Room Availability The Update Room Availability extension point occurs at step 5 of the Basic Flow.

Figure 1. A use case in Jacobson's notation (adopted from [2]).

Use Case: Handle Waiting List
Extension Flows:
EF1. Queue for Room This extension flow occurs at the extension point Update Room Availability in the Reserve Room use case when there are no Rooms of the selected type available.

1. The system creates a pending reservation with a unique identifier for the selected Room type.
2. The system puts the pending reservation into a waiting list.
3. The system displays the unique identifier of the pending reservation to the customer.
4. The base use case terminates.

Figure 2. An extension use case in Jacobson's notation (adopted from [2]).

Use Case: Edit a document
Primary actor: user
Scope: Wapp
Level: user goal
Trigger: User opens the application.
Precondition: none

Main success scenario:

1. User opens a document to edit.
2. User enters and modifies text. . . . . User saves document and exits application.

Use Case: Check spelling Primary actor: user
Scope: Wapp
Level: subfunction!
Precondition A document is open
Trigger: Anytime in **Edit a document** that the document is open and the user selects to run the spell checker.
Main success scenario: . . . etc. . . .

Figure 3. Use cases in Cockburns's notation (adopted from [5]).

plicit extension point either: the extension point is referred to descriptively in the trigger part of the description.

2.4. Tool support

While general UML modeling tools offer some support of use case modeling, there are also dedicated use case modeling tools with usually better coverage of use case description. The challenging areas of use case modeling support are flows and use case relationships because their changes affect the integrity of textual use case descriptions. Based on the creation of textual use case descriptions, it is possible to distinguish three categories of use case tools: text based, template based, and model based tools.

In the text based use case modeling tools, the use case description—including flows and use case relationships—is written as plain or formatted text into respective text boxes. Some tools in this category support formatted text, which makes the textual use case descriptions easier to read. The general problem of these tools is the lack of the support for the use case description maintenance. Examples of such tools include ArgoUML, Poseidon for UML, and IBM Rational Software Architect

In the template based use case modeling tools, a static or dynamic template is used to create use case descriptions. Templates basically partition the description text (plain or formatted). Common partitionings distinguish between flows and a range of simple description items, such as use case name, brief description, preconditions, postconditions, and other similar parts of the use case description that do are not numbered. In case of dynamic template support, the templates can be adapted by adding new or removing existing types of flows and description items to fit the user needs. While the maintainability of use case descriptions is still problematic even in such tools, they are at least easier to read and write. For example, Visual Paradigm and Enterprise Architect fall into this category.

In model based use case modeling tools, use case descriptions are based on a specific use case model. These tools usually contain sophisticated formatted text editors that directly manipulate the structured use case descriptions according to the use case model. For example, every step is a part of a specific flow, which allows the user to perform changes in the order of steps that result in automatic renumbering of affected steps and other parts of the use case description that depend on them like alternative flows or extension points. This is only one of many ways how tools of this category help the user to ensure the integrity of use case descriptions. Visual Use Case and CaseComplete are examples of model based use case modeling tools.

3. Invoking a metamodel

The diversity in use case modeling can be concisely captured in a metamodel, which can serve to better understand this technique and as a basis for the development of configurable use case modeling tools. Since use case modeling is partially covered by the UML metamodel, we will adapt and extend it with the notions needed to cover textual part of use cases. Our aim was not to add to UML specification, but to develop a concise, standalone use case modeling metamodel. Integration into the UML metamodel is a part of our ongoing work.

3.1. Flows

Flows of events (`Flow`) are an essential part of the use case description (see Figure 4). Each flow consists of steps (`Step`). This is a generally accepted idea, yet actual step representation, including their ordering, may vary significantly and as such is beyond this metamodel. However, in our metamodel, we do recognize the possibility of existence of special kinds of steps (`TypedStep`) with the agreed meaning given by their type (`StepType`) which could be defined by its name, the list of parameters and the list of parameter names.

Three types of flows can be recognized: main (basic) flow (`MainFlow`), subflow (`Subflow`), and alternative flow (`AlternativeFlow`). In general, a use case may have any number of main flows—even none—though some approaches require a main flow.

A use case without a main flow would represent a use case that could not be activated directly by an actor. Instead, it would be intended just for inclusion in other use cases or to extend them, in which case it should provide one or more subflows or alternative flows, respectively.

A use case may include preconditions and postconditions, which are a kind of a constraint (`Constraint`). Other parts of the use case description vary significantly among approaches and even in a particular approach depending software analyst preferences, so they are just indicated as any kind of a description item (`DescriptionItem`).

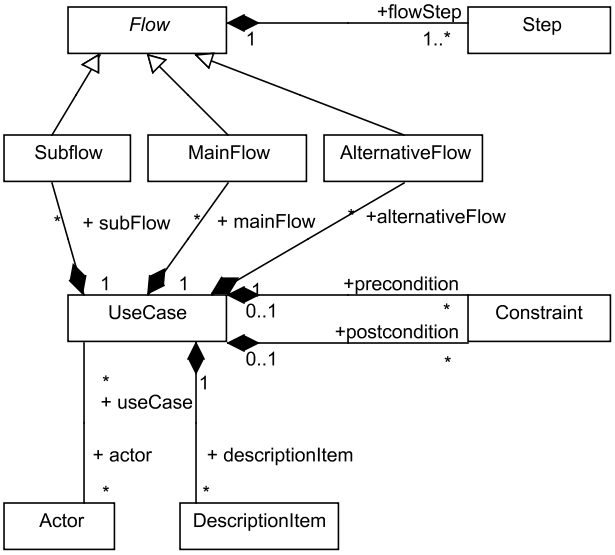

Figure 4. Flows.

In our metamodel, we made the participation relationship between `Actor` and `UseCase` explicit whereas in the UML metamodel it is given by the fact that these two metaclasses are derived from `BehavioredClassifier` which allows for them to be associated.

An alternative flow is activated, usually according to a constraint (`Constraint`), in a particular step (`Step`). In some approaches, it is possible to specify the execution order of the alternative flow with respect to the step affected by it usually before, after, or around it, i.e. with the full control upon the step (just like advices in aspect-oriented programming).

Any flow can have subflows. A use case with no flows can be an abstract use case intended to be specialized [1].

3.2. Relationships

In general, there are two types of use case relationships: include and extend. The UML metamodel recognizes both of them as a special kind of `DirectedRelationship` (which is the metaclass the general dependency is derived from, too, making them a close relative of it). However, some approaches ignore the extend relationship and, hypothetically, there could be approaches that wouldn't provide not even the include relationship.

The include relationship (see Figure 5) means an inclusion of a specific flow from another use case (`FlowInclusion`) in one or several steps of the including use case (`Step`). We opt for inclusion of a general flow

(Flow), although it is unlikely that someone would want to include an alternative flow. The inclusion of a flow may be constrained (Constraint).

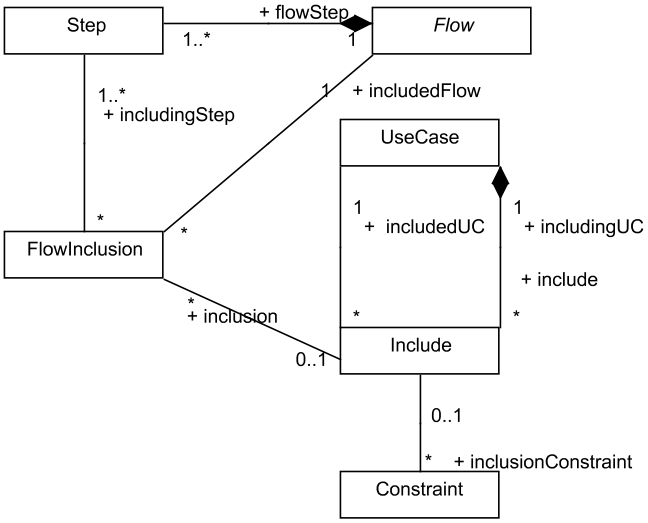

Figure 5. The include relationship.

The inclusion of a flow (FlowInclusion) is possible even without the corresponding include relationship (zero multiplicity of Include), which covers flow activations of use case's own flows.

The extend relationship means an extension of one or several extension points (ExtensionPoint) of the use case being extended by a specific extension flow (FlowExtension). Some approaches allow only an alternative flow to serve as an extending flow, but this is not generally accepted, so our metamodel allows any kind of flow in this role (Fig. 6).

Analogously to alternative flows—which actually act as extension flows in a single use case—some approaches allow to specify the execution order of the extension flow with respect to the extension point, usually before, after, or around it, i.e. with the full control upon the extension point (just like advices in aspect-oriented programming).

An extension point (ExtensionPoint) is merely a name of the step or a range of steps (startingStep–endingStep) represented by an extension location (ExtensionLocation exposed by the use case being extended. The extension of a flow may be constrained (Constraint). In the UML metamodel, there is at most one constraint for each extend relationship. Our metamodel allows several constraints for each extension flow.

As with the flow inclusion, the extension of a flow (FlowExtension) is possible even without the corresponding extend relationship (zero multiplicity of Extend and ExtensionPoint with respect to ExtensionLocation), which covers flow alterations of use case's own flows.

4. Metamodel configuration

The use case modeling metamodel proposed in the previous section can be configured to represent an established notation or simply to define a use case modeling that fits the needs of a particular organization. This can be done mostly by restricting multiplicities of associations in the metamodel. Such a restriction can represent any subset of values allowed by the original multiplicity, but only a total restriction to zero is of practical meaning.

Restricting multiplicities might be seen as a low-level metamodel configuration. To make the configuration easier, we can represent most of the configuration options as Boolean variables where true stands for the original multiplicity, and false for the zero multiplicity. Table 1 presents the list of the most important Boolean configuration options and their values in Jacobson's (J) and Cockburn's (C) notation, which we discussed in Sect. 2.2 and 2.3.

Table 1. The use case metamodel configuration options and their values in Jacobson's (J) and Cockburn's (C) notation.

Property	J	C
Single-Step Extension Points	Y	N
Range Extension Points	Y	N
Mandatory Main Flow	Y	Y
Multiple Main Flows	Y	N
Subflows	Y	Y
Subflows in Main Flows	Y	Y
Subflows in Alternative Flows	Y	Y
Subflows in Subflows	Y	Y
Alternative Flows	Y	Y
Alternative Flows in Main Flows	Y	Y
Alternative Flows in Subflows	Y	Y
Alternative Flows in Alternative Flows	Y	Y
Extension	Y	N
Multiple Extension Locations in an Extension	N	N
Extension by a Specific Flow	Y	N
Extension Flow Constraint	Y	N
Extension Flow Execution Order	Y	N
Inclusion	Y	Y
Inclusion Flow Constraint	N	N
Inclusion of a Specific Flow	N	N

Single-Step Extension Points means the possibility of having startingStep to be equal to endingStep. For *Range Extension Points*, startingStep must be able to differ from endingStep.

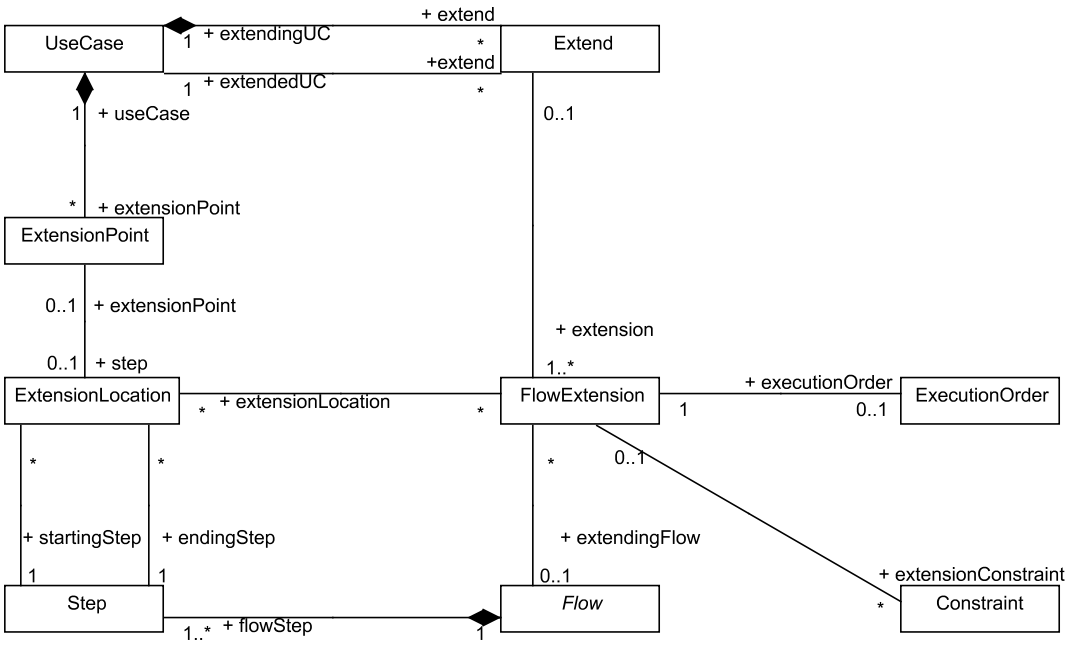

Figure 6. The extend relationship.

Mandatory Main Flow restricts the minimum multiplicity of `mainFlow` to 1, while no *Multiple Main Flows* would mean restricting its maximum to 1.

Options *Subflows*, *Alternative Flows*, *Inclusion*, and *Extension* have a meaning of the very presence of respective metaclasses.

If subflows are allowed, their inclusion can be restricted with respect to the type of the including flow by the following options: *Subflows in Main Flows*, *Subflows in Alternative Flows* and *Subflows in Subflows*. There are analogous options for alternative flows: *Alternative Flows in Main Flows*, *Alternative Flows in Subflows*, and *Alternative Flows in Alternative Flows*. All these options would be realized by constraining associations `Flow–InclusionFlow` and `Flow–ExtensionFlow` to allow only desired subtypes of `Flow`.

The rest of Boolean options has the realization in multiplicity restriction as listed below:

- no *Multiple Extension Locations in an Extension*: maximum `extensionLocation` multiplicity is 1
- no *Extension by a Specific Flow*: maximum `extendingFlow` is 0
- no *Extension Flow Constraint*: maximum `extensionConstraint` is 0
- no *Extension Flow Execution Order*: maximum `executionOrder` is 0
- no *Inclusion Flow Constraint*: maximum `inclusionConstraint` is 0
- no *Inclusion of a Specific Flow*: maximum `includedFlow` is 0

There are two configuration options that consist of a list of elements (not listed in the table). *Description Items* represents a (possibly) empty list of textual description items a use case can have. *Execution Order Types* is a list of values that represent possible execution order types of extension flows, usually before, after, or around (as has been mentioned in Sect. 3.1).

The presence of options *Alternative Flows in Subflows*, *Alternative Flows in Alternative Flows*, and *Multiple Extension Locations in an Extension* in Jacobson's notation configuration is estimated: the notation allows for these options, but we haven't actually encountered examples that would employ them.

5. Evaluation

In order to evaluate our approach, we have developed a prototype of a configurable use case modeling tool. The tool supports the main, textual part of use case modeling.

We were mainly interested in testing of the metamodel and choice of configuration options in practice. We identified a possibility of having inconsistent configurations of options. Consider a configuration with selected option *Subflows* and *Subflows in Alternative Flows*, but with no *Alternative Flows*, nor *Alternative Flows in Main Flows*. Of

course, to solve this, it would be sufficient to include the *Alternative Flows* and *Alternative Flows in Main Flows* options.

Although in the current tool implementation this problem does not actually produce inconsistent use case models, users may be confused by not having the actual capabilities of the options they selected available. This is actually a feature interaction problem. To deal with them, feature modeling as an appropriate approach to configuration representation and validation could be used [7, 21].

6. Related Work

Hoffman et al. [19] recently[1] proposed an extension of the UML metamodel to support textual use case description. While our aim was to enable precise definition of different use case notations to enable their consistent application, Hoffman et al. are concerned mainly with ensuring consistency between use case diagrams and descriptions. To achieve this, they include the steps in the use case flows in their metamodel. This is complementary with respect to the metamodel proposed in this paper. However, in our metamodel, it would be necessary to support notational variants of step representation.

The UML metamodel [14] is a prominent attempt of establishing a common diagrammatical use case modeling notation, but it also defines a set of notions, which we find useful as a basis for our use case modeling metamodel.

Rui and Butler [17] proposed a use case modeling metamodel focused on a single use case modeling notation. Others have focused on unifying specific notational issues in use case modeling such as alternative flow types [11], formalizing the include and extend relationships [3], or even formalizing use cases as such [18].

While the use case modeling metamodel proposed in this paper also attempts to cover diversity of options in use case modeling, its goal is not to unify them—at least not more than necessary—but to map the common and variable among them and to provide a way to opt for a particular notation or a combination of several notations as needed. The latter requires a consistency check, which the proposed metamodel is capable of, too.

7. Conclusions and Further Work

In this paper we proposed a use case modeling metamodel. The metamodel is based on UML metamodel and embraces mainly Jacobson's and Cockburn's use case notation.

The proposed metamodel is configurable by including or omitting some of its elements, posing some constraints on their use, or by constraining multiplicity of relationships among them. Its configurations represent individual use case modeling notations, so it can be used to formally define the syntax of a particular use case modeling notation that should be followed manually in general modeling tools or enforced by a dedicated tool.

We expect our metamodel would develop further to embrace other possibilities of use case modeling. For example, we have not considered explicitly business use cases [9], which certainly deserve attention. We plan to extend our configurable use case modeling tool prototype and perform experiments that embrace new features.

With respect to configuration and feature interaction problems, we would like to explore the possibility of employing feature modeling [7, 21]. For getting a full use of a feature model, the metamodel would have to be transformed to include all the possibilities of its configuration space and have them mapped to respective features according to the approach of the superimposed variants [6]. While feature modeling is basically in accordance with the graphical way of expressing metamodels we stick to, some constraints have to be expressed in a non-graphical form [21]. Other object-oriented metamodels have been expressed in a purely non-graphical form [16] even in cases when they define graphical models [13].

Yet another line of further work is to improve the integration of the elements of our use case modeling metamodel with the UML metamodel.

Acknowledgements The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09.

References

[1] J. Arlow and I. Neustadt. *UML 2 and the Unified Process*. Addison-Wesley, 2005.

[2] K. Bittner and I. Spence. *Use Case Modeling*. Addison-Wesley, 2002.

[3] A. Bragança and R. J. Machado. Extending uml 2.0 metamodel for complementary usages of the «extend» relationship within use case variability specification. In *Proc. of 10th International Software Product Line Conference, SPLC 2006*, pages 123–130, Baltimore, USA, 2006. IEEE Computer Society Press.

[4] R. J. A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, 1998.

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

[6] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Glück and M. R. Lowry, editors, *Proc. of Generative Programming and Component Engineering, 4th Inter-*

[1] We became aware of this work during the final editing of our paper.

national Conference, GPCE 2005, LNCS 3676, pages 422–437, Tallinn, Estonia, Oct. 2005. Springer.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[8] A. Dedeke and B. Lieberman. Qualifying use case diagram associations. *IEEE Computer*, 39(6):23–29, June 2006.

[9] J. Heumann. Introduction to business modeling using the unified modeling language (uml). developerWorks, IBM, Nov. 2003. `http://www.ibm.com/developerworks/rational/library/360.html`.

[10] I. Jacobson and N. Pan-Wei. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.

[11] P. Metz, J. O'Brien, and W. Weber. Specifying use case interaction: Types of alternative courses. *Journal of Object-Oriented Programming*, 2(2):111–131, Mar. 2003.

[12] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[13] M. Navarčik and I. Polášek. Object model notation. In *Proc. of 8th International Conference on Information Systems Implementation and Modelling, ISIM 2005*, Rožnov pod Radhoštěm, Czech Republic, 2005.

[14] Object Management Group. OMG unified modeling language (OMG UML), superstructure, v2.1.2, Nov. 2007. `http://www.omg.org/docs/formal/07-11-02.pdf`.

[15] T. Pender. *UML Bible*. Wiley, 2003.

[16] J. Porubän and P. Václavík. Generating software language parser from domain classes. In *Proc. of International Scientific Conference on Computer Science and Engineering, CSE 2008*, pages 133–140, Stará Lesná, Slovakia, Sept. 2008.

[17] K. Rui and G. Butler. Refactoring use case models: The metamodel. In M. J. Oudshoorn, editor, *Proc. of 26th Australasian Computer Science Conference, ACSC 2003*, pages 301–308, Adelaide, Australia, Feb. 2003.

[18] P. Stevens. On use cases and their relationships in the unified modelling language. In H. Hußmann, editor, *4th International Conference on Fundamental Approaches to Software Engineering, FASE 2001, held as a part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001*, LNCS 2029, pages 140–155, Genova, Italy, Apr. 2001. Springer.

[19] A. N. Veit Hoffmann, Horst Lichter. Towards the integration of uml- and textual use case modeling. *Journal of Object Technology*, 8(3):85–100, 2009. `http://www.jot.fm/issues/issue_2009_05/article3/`.

[20] G. Övergaard and K. Palmkvist. *Use Cases: Patterns and Blueprints*. Addison-Wesley, 2004.

[21] V. Vranić. Reconciling feature modeling: A feature modeling metamodel. In M. Weske and P. Liggsmeyer, editors, *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, LNCS 3263, pages 122–137, Erfurt, Germany, Sept. 2004. Springer.