Noname manuscript No.
(will be inserted by the editor)

A Configurable Use Case Modeling Metamodel with Superimposed Variants

Valentino Vranić · Ľuboš Zelinka

Received: 15 December 2009 / Accepted: 18 April 2010

Abstract There is a variety of approaches to use case modeling, especially regarding textual use case description as their true form. Under certain circumstances, the use of each one of these approaches may be justified. 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 adapts and extends UML metamodel elements relevant to use cases that covers their graphical portion to cover different use case modeling notations with a special attention given to the elements of textual expression of flows of events in use cases. The configuration options of the proposed use case modeling metamodel and its configurations representing Jacobson's and Cockburn's notation are presented and discussed. To better express configuration dependencies and avoid option interaction (due to which an unexpected behavior occurs), revealed in a practical evaluation by a configurable use case modeling tool prototype, the options have been arranged into a feature model and the approach of superimposed variants has been applied to the metamodel. The metamodel may serve as a basis for a configurable use case modeling tool or notation-specific tools. More important, it provides a framework for a consistent application of the use case modeling notation in one or across several organizations. It can also be used to facilitate a use case model interchange between notation-specific tools based on the metamodel.

V. Vranić · Ľ. Zelinka
Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Ilkovičova, 84216 Bratislava 4, Slovakia
E-mail: vranic@fiit.stuba.sk

Keywords Use case · Metamodel · Configuration · Feature model · Superimposed variants

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 use case diagrams. Indeed, although there are other graphical notations of use case diagrams ranging from those close to the UML notation [6, 10] to some quite different notations [5], in practice, use case diagrams are truly a realm of UML. However, UML does not prescribe any notation for the textual use case description, which is the main form of use cases as they were invented [13], while use case diagrams serve mostly as an overview or catalog of the use case model [6].

There is a variety of approaches to use case modeling, especially regarding textual use case description. Under certain circumstances, the use of each one of these approaches may be justified. Developers are interested in having their use case modeling notation 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 models that may exist within a notation—i.e., a metamodel—is what is actually needed.

Besides having a separate metamodel of each notation, a metamodel could cover several sufficiently related notations and be configured to describe each one of them as needed. It appears that use case modeling notations are close enough to each other to allow for constructing such a common, configurable use case modeling metamodel, which is the topic of this paper.[1]

The use case modeling metamodel proposed here aims at covering the existing variety of approaches to use case modeling, especially regarding their textual description as their true form. Such a metamodel may serve as a basis of a configurable use case modeling tool. Its configurations represent metamodels of individual use case modeling notations and may be employed in notation-specific tools, but, what is more important, having a metamodel of a specific notation provides also a framework for its consistent application in one or across several organizations.

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, Section 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 5 describes how the options have been arranged into a feature model and the approach of superimposed variants has been applied to the metamodel in order to better express configuration dependencies and avoid option interaction revealed in a practical evaluation by a configurable use case modeling tool prototype, Section 6 explains the possibilities of the metamodel application. Section 7 discusses related work. Section 8 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 [2]. 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 textual use case description as such, and then points out the differences between Jacobson's and Cockburn's notation. The way use case modeling 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.

[1] This paper is an extended version of the paper presented at ECBS-EERC 2009 [25]. Some findings presented there have been revised and the configuration of the metamodel has been supported by a feature model and the approach of superimposed variants.

2.1 Textual Use Case Description

A semiformal textual use case description is simply a natural language text structured using a 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. The common parts of the textual use case description are discussed in this section.

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 [2]. 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 [17], but we may encounter different, less restrictive understanding of preconditions and postcondition, putting them to a merely informative position [26], or a fully restricted view, where the very use case activation is presumed by fulfilling its precondition [2]. Preconditions and postconditions are expressed in the form of natural language statements.

Flows of events—or simply just *flows* (known also as *scenarios*)— represent every possible outcome of an attempt to accomplish a use case goal [20]. 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 [14]. In the textual 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 [6], as some steps simply do not 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 [3]. 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 [3]. 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 [14].

Use case relationships are a part of the textual 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 [19]. 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 functional decomposition. Functional decomposition of use cases is in general considered bad practice [6, 2] because it leads to a model with high-level use cases that do not represent real usage scenarios with a clear goal, but just an artificial structuring of requirements. For example, an inexperienced use case modeler may be tempted to identify a use case like *Process Student Data* that would include all kinds of processing student date, but would not provide a goal of such processing.

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 [19]. 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 [3]. Every flow of events can have multiple extension points. A common way to define an extension point in a textual use case description is inserting its name into the flow of events between two steps. It is a pretty straightforward way, but every such extension point refers only to a single step in the flow of events, which can be limiting. Since steps in flows of events are usually numbered, it is possible to define an extension point by a step number or as a range of steps.

2.2 Jacobson's Notation

Jacobson's and Cockburn's use case modeling notation are well established and distinguished. Consider the example in Figures 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 suggests 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 [14].

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 Figure 3, the *Check spelling* use case extends the *Edit a document* use case implicitly by its main flow. There is no explicit extension point either: the extension point is referred to descriptively in the trigger part of the textual description.

2.4 Tool Support

While general UML modeling tools such as IBM Rational Software Modeler or Enterprise Architect offer some support of use case modeling, there are also dedicated use case modeling tools with usually better coverage of textual 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 textual use case description—including flows and use case relationships—is written as plain or formatted text into an unstructured text box. 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 textual 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 textual use case descriptions. Templates basically partition the description text (plain or formatted). Common partitionings distinguish between flows and a range of sim-

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.

B2. Reserve Room by Phone

1. The receptionist receives a call or fax from a customer wanting to reserve a room.

...

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.

Fig. 1 A use case in Jacobson's notation [14]; adapted.

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.

Fig. 2 An extension use case in Jacobson's notation [14]; adapted.

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:

. . .

Fig. 3 Cockburns's use case modeling notation [6].

ple description items, such as use case name, brief description, preconditions, postconditions, and other similar parts of the textual use case description that 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 textual 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, textual use case descriptions are based on a specific use case model. These tools usually contain sophisticated formatted text editors that directly manipulate the structured textual 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 textual 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 textual use case descriptions. Visual Use Case and CaseComplete are examples of model based use case modeling tools.

3 Establishing a Metamodel

The diversity in use case modeling can be concisely captured in a configurable metamodel, which can serve as a basis for the development of configurable use case modeling tools. Each configuration of the overall metamodel represents the metamodel of a specific use case modeling notation and can be used to regularize the application of this notation similarly as the UML metamodel regularizes the application of UML.

Since use case modeling is partially covered by the UML metamodel, we will adapt and extend it with the notions needed to cover the textual part of use cases. However, our aim was not to extend the UML specification, but to develop a concise, standalone use case modeling metamodel. Integration into the UML metamodel is a part of our ongoing work.

As the UML metamodel, we also rely on the notion of a metaclass that represents a class of the notation element. The notation elements, in turn, are classes themselves; hence the name metaclass, which literally means a class beyond another class.

3.1 Flows

Flows of events (*Flow*) are an essential part of the textual 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 specialized steps (*TypedStep*) with an agreed meaning given by their type (*StepType*) which could be defined by their name, the list of parameters, and the list of parameter names. Examples of step types embrace various conditional and loop statements.

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 may require a main flow [6].

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 textual use case description vary significantly among approaches and even in a particular approach depending on software analyst preferences, so they are just indicated as any kind of a description item (*DescriptionItem*).

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 [19].

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 [14].

Any flow can have subflows. A use case with no flows can be an abstract use case intended to be specialized [2].

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 [19]. However, some organizations tend to ignore the extend relationship and, hypothetically, there could be organizations that wouldn not 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*).

The inclusion of a flow (*FlowInclusion*) is possible even without the corresponding include relationship (zero multiplicity of *Include*), which covers flow activations of a use case 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 [14], but this is not

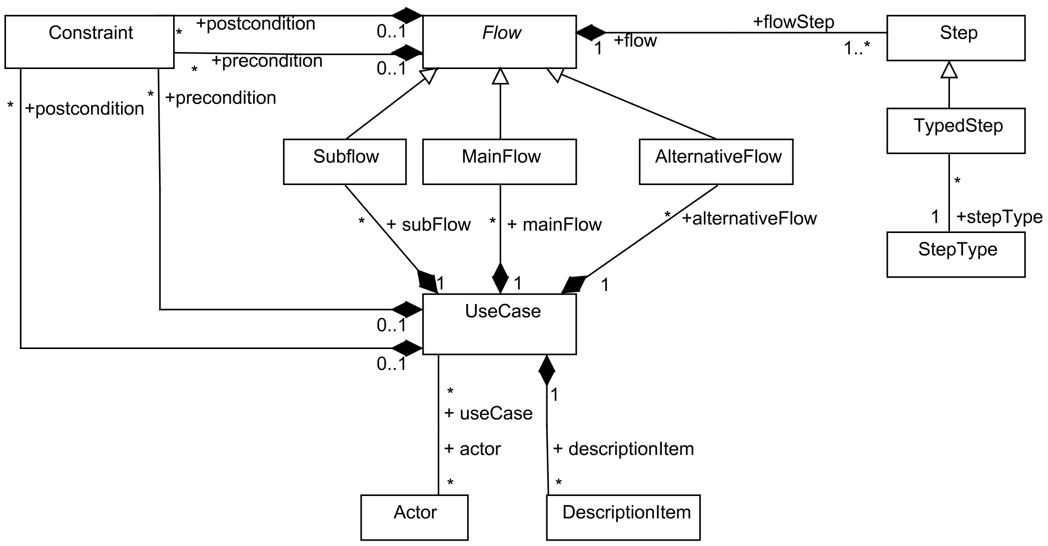


Fig. 4 Flows.

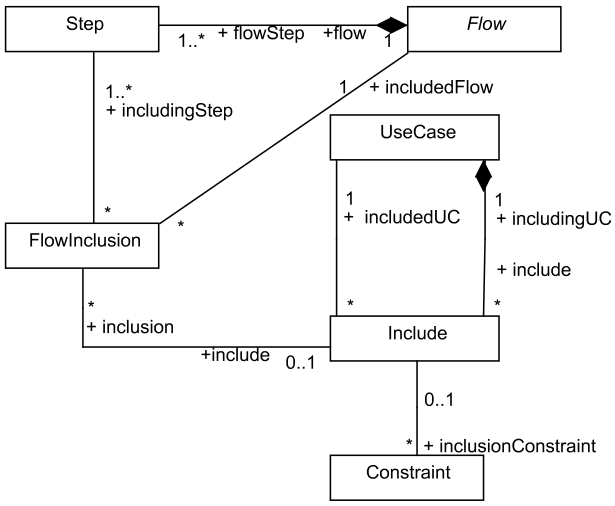


Fig. 5 The include relationship.

generally accepted, so our metamodel allows any kind of flow in this role (Figure 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, again usually before, after, or around it, i.e. with the full control upon the extension point.

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 a use case own flows.

4 Basic 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 directly omitting metaclasses or 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 for the metamodel configuration presented here since it is equal to the omission of the respective association. Also, if multiplicities of all roles a metaclass participates in are restricted to zero, it can be omitted from the metamodel.

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

Single-Step Extension Points requires the lower limit of the multiplicity of the *endingStep* role of the *Step* metaclass to be 0. *Range Extension Points* requires the

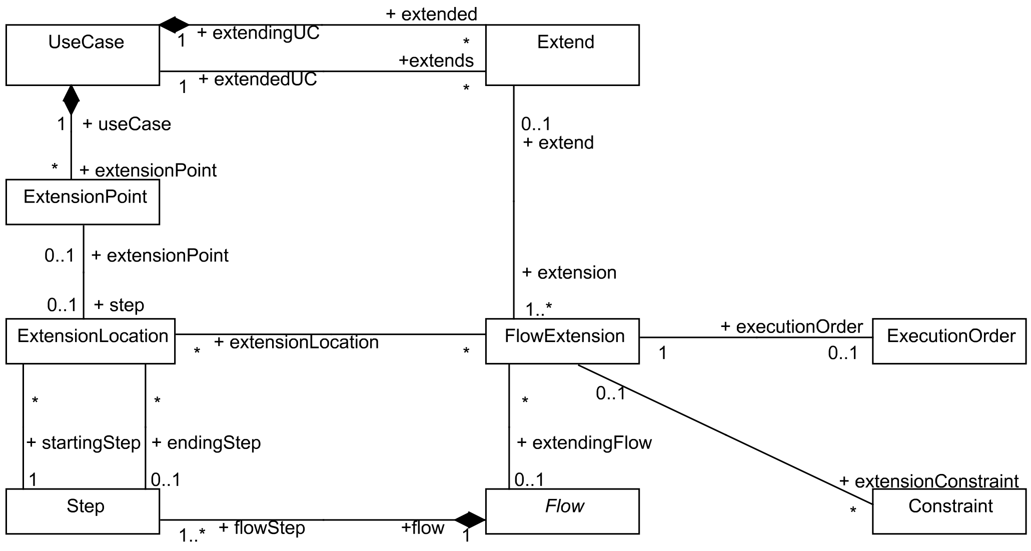


Fig. 6 The extend relationship.

Table 1 The use case metamodel configuration options and their values (Y if present, N if not present) 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	N	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	Y
Multiple Extension Locations in an Extension	N	N
Extension by a Specific Flow	Y	Y
Extension Constraint	Y	N
Extension Flow Execution Order	Y	N
Inclusion	Y	Y
Inclusion Constraint	N	N
Inclusion of a Specific Flow	N	N
Description	Y	Y
Typed Steps	N	N
Preconditions	Y	Y
Postconditions	Y	Y
Flow Preconditions	N	N
Flow Postconditions	N	N

upper limit of the multiplicity of the *startingStep* role of the *Step* metaclass to be `1` (see Figure 6).

Mandatory Main Flow restricts the minimum multiplicity of the *mainFlow* role of the *MainFlow* metaclass to `1`, while no *Multiple Main Flows* would mean restricting its maximum to `0` (see Figure 4).

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

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 with respect to what types of flows they can be applied: *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 are realized by restricting the corresponding maximum multiplicities of the roles as listed below (the metaclasses to which the roles apply are introduced in parentheses):

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

The *Description* option has the meaning of the presence of the *Description Item* metaclass (see Figure 4).

The *Typed Steps* option specifies whether the notation provides step types, i.e. whether the *TypedStep* and *StepType* metaclasses are present. The notation could be configured further by providing specific values allowed for the instances of the *DescriptionItem* metaclass to determine the structure of the textual use case description. Similarly, the instances of the *TypedStep* and *StepType* metaclasses could be specified to determine step types. Also, instances of the *ExecutionOrder* metaclass could be used to specify possible execution order types of extension flows, which are usually before, after, and around (as has been mentioned in Section 3.1).

The absence of the *Preconditions* and *Postconditions* option limits the maximum multiplicity of the *precondition* and *postcondition* of the *Constraint* metaclass roles to `0`. Similarly, the absence of the *Flow Preconditions* and *Flow Postconditions* option limits the maximum multiplicity of the *precondition* and *postcondition* roles of the *Constraint* metaclass to `0`.

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

5 Metamodel Configuration Based on Superimposed Variants

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 it, feature modeling as an appropriate approach to configuration representation and validation could be used [8,27]. In this section, a feature model of the use case modeling notation is presented and the use case modeling metamodel is annotated with features from this model that determine the presence of its elements.

5.1 The Feature Model of the Use Case Modeling Notation

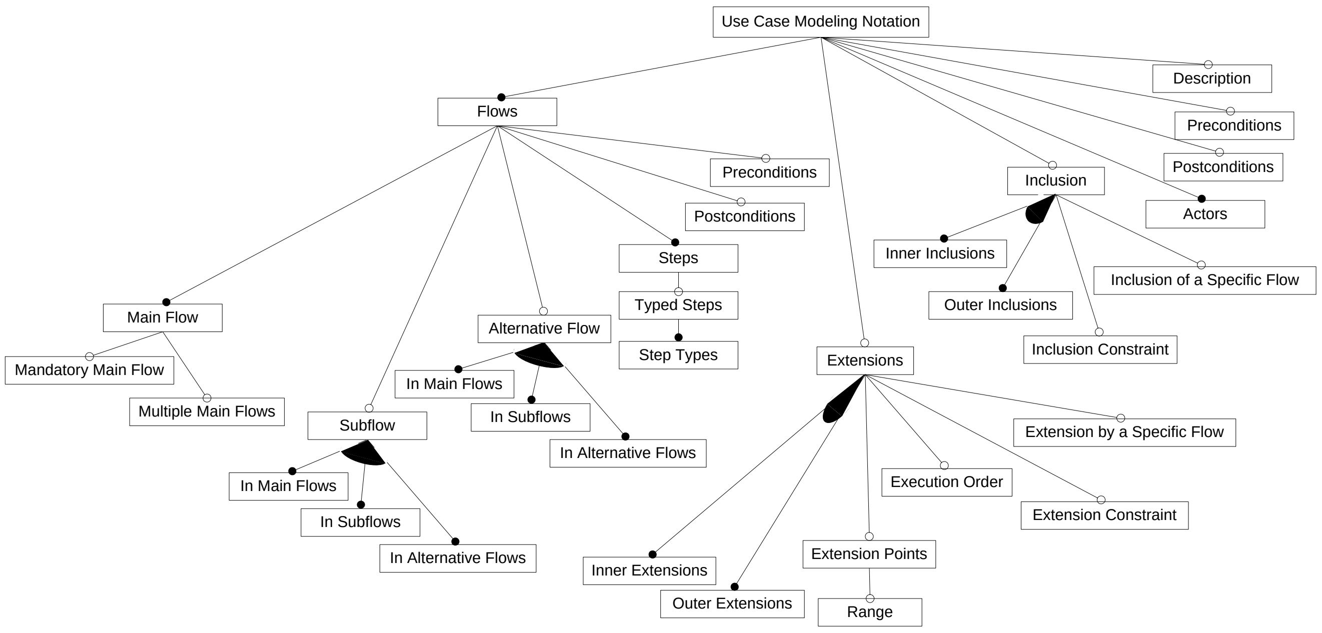
Feature modeling is a conceptual domain modeling technique in which concepts are expressed by their features taking into account feature interdependencies and variability in order to capture the concept configurability [27,8]. It is used to express configurability abstractly in software product lines shaping their whole implementation.

There are several feature modeling notations which are in practice often extended to cover project specific issues. We will use the basic Czarnecki–Eisenecker feature modeling notation which is based on the original FODA notation [15].

Feature diagrams are the main part of feature models. Feature diagrams are directed trees whose root represents a concept with the rest of the nodes being its features. Variability of the features is expressed by different kinds of edges and edge decorations (arcs) and by the structure of the tree since a feature may be selected only if its parent feature has been selected. Textually represented additional constraints may accompany feature diagrams.

Feature configurations that are in accordance with the feature variability constraints represent concept instances [27]. They may be represented also by feature diagrams. If all configuration is static, then concept instances will contain no variable features. We do not consider here more sophisticated approaches to concept instantiation like the staged configuration [9] or instantiation time based approach [27].

The concept that has to be modeled here is the use case modeling notation. As features in feature modeling are understood as important properties of concepts intended primarily to discriminate between concept instances [8], Table 1 represents a good source of features for this concept. Figure 7 shows the feature model of the use case modeling notation. All use case modeling notations have the notion of a flow, so it is modeled as a mandatory feature *Flow*, which is denoted by an edge with a filled circle arrowhead. There is always possibility to have a main flow, which is modeled as a mandatory subfeature of the *Flow* feature. Actually, the main flow may be mandatory in all use cases, which is expressed by the optional *Mandatory Main Flow* feature. Optionality is denoted by an empty circle arrowhead. Also, there may be multiple main flows in the use case, which is also modeled as an optional feature, *Multiple Main Flows*.



1. *Use Case Modeling Notation.Flows.Flow Types.Main Flow.Mandatory Main Flow* ⇒ (*Use Case Modeling Notation.Flows.Flow Types.Subflow* ∨ *Use Case Modeling Notation.Flows.Flow Types.Alternative Flow*)
2. *Inner Extensions* ⇔ *Alternative Flow*
3. *Inner Inclusions* ⇒ *Inclusion of a Specific Flow*
4. *Inner Extensions* ⇒ *Extension of a Specific Flow*

Fig. 7 The feature model of the use case modeling notation.

Not all use case modeling notations have to support subflows and alternative flows, whose availability is therefore modeled by optional features *Subflow* and *Alternative Flow*. Notations may restrict the applicability of subflows with respect to the type of the flow from which they may be referred to. This is modeled by a group of or-features (indicated by a filled arc) consisting of the following features: *In Main Flows*, *In Subflows*, and *In Alternative Flows*. At least one feature has to be selected in the group of or-features. By this, a meaningless configuration with an unspecified subflow application would be prohibited.

If the notation supports neither subflows, nor alternative flows, there is no sense in stating explicitly that a main flow is mandatory. This is expressed by constraint 1 in Figure 7. Constraints are expressed as predicate logic expressions with only one predicate which takes a feature as its only parameter and evaluates to true if the feature is included in the current concept instance. To keep the expressions concise, the actual predicate is omitted and only feature names are displayed. Furthermore, the name of the feature is unqualified if it is unique in the whole feature model.

The same scheme applies to alternative flows with the difference in the interpretation of the or-features modeling their applicability. In this case, due to the nature of alternative flows, they represent types of the flows that may be affected by alternative flows.

In all notations, flows consist of steps, which is expressed by the *Steps* mandatory feature. Some notations may support typed steps, modeled by the optional *Typed Step* feature, which, in turn, requires step types to be defined, expressed by the mandatory *Step Types* feature.

A notation may allow preconditions and postconditions to be applied to specific flows, which is modeled by the *Preconditions* and *Postconditions* optional subfeatures of the *Flow* feature. Some notations may allow preconditions and postconditions to be associated with use cases as such, which is modeled by equally named direct features of the *Use Case Modeling Notation*. The two possibilities are not exclusive.

Apart from flows, a use case itself may be provided with a textual description, which is modeled by the *Description* optional feature. All notations require actors, so they are modeled by the *Actors* mandatory feature.

A notation may prohibit the use of the include relationship between use cases since this often leads to the undesired functional decomposition (see Section 2.1), so its presence is modeled by the *Inclusions* optional feature. The include relationship may be targeted not only at a use case as such, but also at a specific flow, which is modeled by the *Inclusion of a Specific Flow* optional feature. This flow may be a flow of the including use case itself, which will be denoted as an inner inclusion, or a flow of another use case, which will be denoted as external inclusion. Obviously, inner inclusions may be realized only by specific flows and this is expressed by constraint 3 in Figure 7. Also, some notations may support constraining the include relationship and this is modeled by the *Inclusion Constraint* optional feature. The notation may support inner inclusions (inside of one use case) and outer inclusions with the two not being exclusive and at least one having to be selected, which is expressed by the *Inner Inclusions* and *Outer Inclusions* or-features.

The extend relationship also may or may not be supported by a notation, and this is modeled by the *Extensions* optional feature. As the include relationship, it may also be realized at the flow level or not, modeled by the *Extension by a Specific Flow* optional feature, with some notations supporting constraining it, modeled by the *Extension Constraint* optional feature. Similarly to inclusions, extensions may also be inner and outer, which is expressed by the *Inner Extensions* and *Outer Extensions* or-features. Inner extensions are in fact alternative flows, and this is expressed by constraint 2 in Figure 7. Similarly as inner inclusions, inner extensions—which are actually alternative flows—may be realized only by specific flows and this is expressed by constraint 4 in Figure 7. Some notations may allow to express the order in which an extension is to be executed with respect to the affected extension point, which is captured by the *Extension Order* optional feature. Extension points may be employed, which is modeled by the *Extension Points* optional feature. If they are employed, the notation may support expressing the range of extension points to be affected by an extension, which is modeled by the *Range* optional subfeature of the *Extension Points* feature.

5.2 Superimposed Use Case Modeling Metamodel

The approach of superimposed variants can be successfully used to automate model configuration [7]. A model contains all possible variants superimposed. The configuration space is represented by a feature model. The elements of the model—or actually template model—are annotated with so-called *presence conditions* and *metaexpressions*[2] defined in terms of features. Presence conditions determine the presence of the model elements they annotate. Metaexpressions are used to determine some properties of model elements (e.g., their names or types they operate on). By this, the selection of features fully determines which model elements and relationships among them should be present. The automation of this process can be easily achieved by a simple tool [1].

The basic idea of the approach is illustrated by Figure 8. A simple UML class model[3] with two superimposed variants is configured by a feature model. Presence conditions are introduced in the form of tagged values (in curly brackets). Class *A* is present in the model only if feature *F5* is selected. Similarly, class *B* is present in the model only if feature *F6* is selected. Note that associations are not annotated: they are removed by implicit presence conditions as their presence is determined by the presence of the classes they connect. The implicit presence conditions for class diagrams are quite intuitive; formally, they have been proposed by Czarnecki and Antkiewicz [7].

The use case modeling metamodel presented in Section 3 is already a template model as required by the approach of superimposed variants, so it will be sufficient to annotate it with the presence conditions and metaexpressions. Figure 9 shows the annotated part of the metamodel describing flows. As in the introductory example to superimposed variants (Figure 8), presence conditions are introduced in the form of tagged values. The tag is denoted as *config* and the value is the name of the feature that determines the presence of the annotated element.

Multiplicities are not modified directly, but constrained using OCL invariants. For configuration purposes, we assume the existence of the *Config* class with the operation *inconfig()* taking one string parameter and returning a Boolean value. The operation returns true if the feature model configuration contains a feature with the name provided as a parameter. As with feature model additional constraints, the name of the feature is unqualified if it is unique in the whole feature model. As an example, consider the first OCL constraint in the group of constraints attached to *Use Case* in Figure 9: if the notation requires a mandatory main flow in each use case, that restricts *Use Case* to *MainFlow* multiplicity at the *mainFlow* role to be at least one.

Figure 10 shows the annotated part of the metamodel describing the include relationship. The OCL constraint attached to *FlowInclusion* (indicated by ellipsis) due to its length is introduced here:

```
inv: not Config::inconfig("Inner Inclusions")
        implies include->count() = 1
```

[2] originally *meta-expressions [7]*

[3] one of basic object-oriented models that describes classes and relationships among them

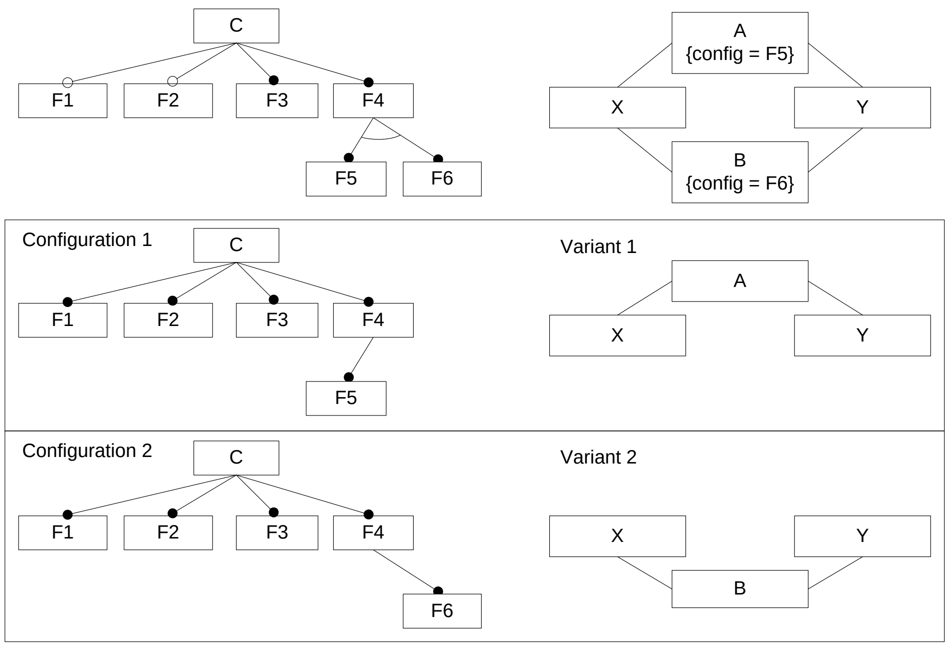


Fig. 8 Configuring a model containing superimposed variants.

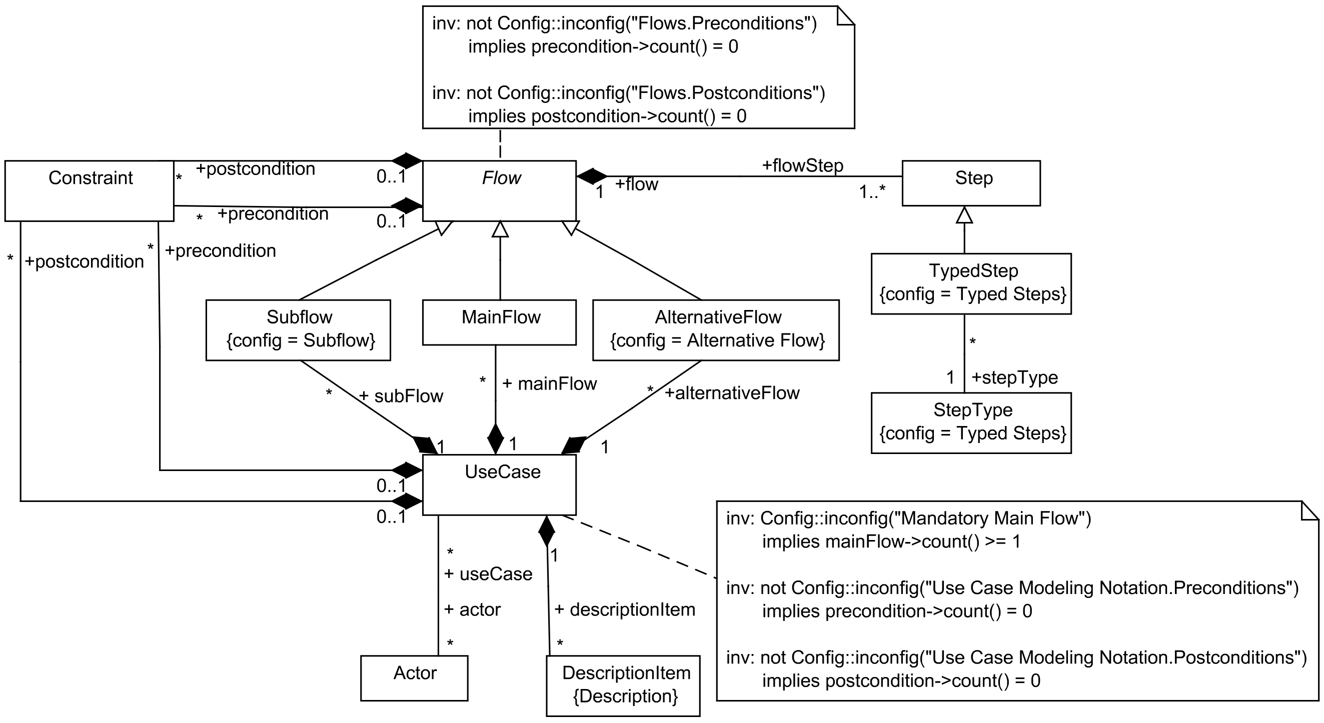
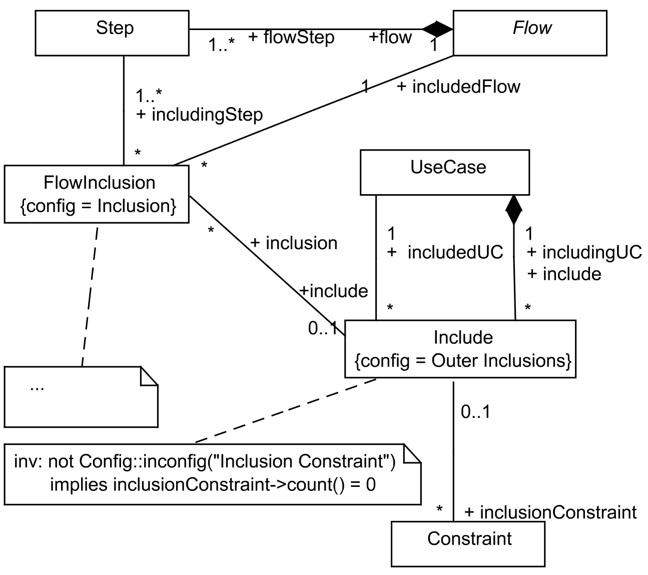

inv: not Config::inconfig("Flows.Preconditions") implies precondition->count() = 0
inv: not Config::inconfig("Flows.Postconditions") implies postcondition->count() = 0
Constraint
Flow
Step
+postcondition
+precondition
+flowStep
+flow
1..*
0..1
TypedStep {config = Typed Steps}
StepType {config = Typed Steps}
+stepType
Subflow {config = Subflow}
MainFlow
AlternativeFlow {config = Alternative Flow}
+ subFlow
+ mainFlow
+alternativeFlow
UseCase
+ useCase
+ actor
+ descriptionItem
Actor
DescriptionItem {Description}
inv: Config::inconfig("Mandatory Main Flow") implies mainFlow->count() >= 1
inv: not Config::inconfig("Use Case Modeling Notation.Preconditions") implies precondition->count() = 0
inv: not Config::inconfig("Use Case Modeling Notation.Postconditions") implies postcondition->count() = 0

Fig. 9 The flow metamodel part annotated with features.

```
inv: includedFlow.oclIsTypeOf(Subflow)
     and includingStep.flow−>forAll(
        o : Flow | o.oclIsTypeOf(MainFlow))
     implies Config::inconfig("
        Use Case Modeling Notation.
        Subflow.In Main Flows")

inv: includedFlow.oclIsTypeOf(Subflow)
     and includingStep.flow−>forAll(
        o : Flow | o.oclIsTypeOf(Subflow))
     implies Config::inconfig("
        Use Case Modeling Notation.
        Subflow.In Subflows")

inv: includedFlow.oclIsTypeOf(Subflow)
     and includingStep.flow−>forAll(
        o : Flow | o.oclIsTypeOf(AlternativeFlow))
     implies Config::inconfig("
        Use Case Modeling Notation.
        Subflow.In Alternative Flows")
```



Step
Flow
1..* + flowStep
+flow 1
1 + includedFlow
1..* + includingStep
*
FlowInclusion {config = Inclusion}
*
UseCase
* + inclusion
+include
0..1
1 + includedUC
1 + includingUC + include
*
*
Include {config = Outer Inclusions}
...
0..1
inv: not Config::inconfig("Inclusion Constraint") implies inclusionConstraint->count() = 0
* + inclusionConstraint
Constraint

Fig. 10 The include relationship metamodel part annotated with features.

The first invariant attached to the *FlowInclusion* metaclass ensures that if the notation does not support inner inclusions, each flow inclusion is bound to an include relationship. The rest of the invariants attached to the *FlowInclusion* metaclass serve to transfer the limitations of the applicability of subflows with respect to the flow types that employ them given by the feature model to the metamodel.

Figure 11 shows the annotated part of the metamodel describing the extend relationship. The OCL constraint attached to *FlowInclusion* (indicated by ellipsis) due to its length is introduced here:

```
inv: not Config::inconfig("Inner Extensions")
        implies extend−>count() = 1

inv: not Config::inconfig("Extension Constraint")
        implies extensionConstraint−>count() = 0

inv: extendingFlow.oclIsTypeOf(Alternative Flow)
        and extensionLocation.startingStep.flow−>forAll(
            o : Flow | o.oclIsTypeOf(MainFlow))
        implies Config::inconfig("
            Use Case Modeling Notation.
            Alternative Flow.In Main Flows")

inv: extendingFlow.oclIsTypeOf(Alternative Flow)
        and extensionLocation.startingStep.flow−>forAll(
            o : Flow | o.oclIsTypeOf(Subflow))
        implies Config::inconfig("
            Use Case Modeling Notation.
            Alternative Flow.In Subflows")

inv: extendingFlow.oclIsTypeOf(Alternative Flow)
        and extensionLocation.startingStep.flow−>forAll(
            o : Flow | o.oclIsTypeOf(AlternativeFlow))
        implies Config::inconfig("
            Use Case Modeling Notation.
            Alternative Flow.In Alternative Flows")
```

The first invariant attached to the *FlowExtension* metaclass ensures that if the notation does not support inner extensions, each flow extension is bound to an extend relationship. The second invariant regulates the presence of extension constraints. The rest of the invariants attached to the *FlowExtension* metaclass—similarly to those attached to the *FlowInclusion* metaclass—serve to transfer the limitations of the applicability of alternative flows with respect to the affected flow types given by the feature model to the metamodel.

6 Applying the Metamodel

The metamodel may serve as a basis for a configurable use case modeling tool. As we mentioned in Section 5, we developed a prototype of such a tool that directed us towards feature modeling as an appropriate way of expressing nontrivial configuration options. In a configurable use case modeling tool, the metamodel with superimposed variants would have to be configured dynamically according to these options.

Apart from this, the metamodel can be used to assure notation consistency or to develop notation-specific tools. It also provides a basis for the use case model interchange among such tools.

6.1 Assuring Notation Consistency

Desired properties of the notation can be conveniently expressed in terms of the use case modeling notation feature model. A particular notation metamodel can be derived from the general metamodel according to the features that have been selected by their manual or automatic (requiring an additional tool support) evaluation. Such a configured metamodel could be used as a basis for a notation-specific tool. More important, it would provide a framework for a consistent application of the notation in one or across several organizations.

Consider, for example, an organization decides to enforce a consistent application of a particular use case notation by providing their analysts with this notation metamodel. Suppose they opt for a simpler notation with at most one main flow per use case and with alternative flows, but no subflows (see Figure 12). Each

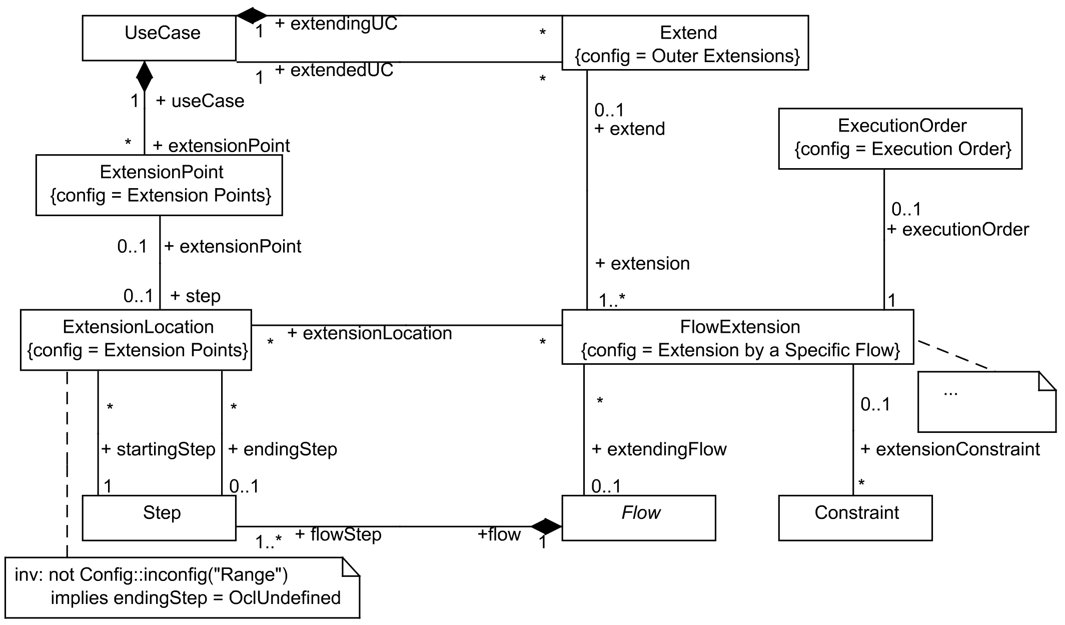


Fig. 11 The extend relationship metamodel part annotated with features.

flow may be accompanied with its own preconditions and postconditions. Use cases as such have no preconditions or postconditions. The notation supports both outer and inner inclusions and extensions with a possibility to include or extend by a specific flow. It also supports extension constraints and relies on a descriptive expression of extension points.

Figure 13 shows the resulting configuration of the use case modeling metamodel. The OCL constraints in the metamodel have been evaluated statically where possible to make as much as possible information available directly in the diagram. This resulted in constraining the *UseCase–MainFlow* multiplicity at *mainFlow* to `1..*` (was `*`), constraining the *UseCase–Constraint* multiplicity at *precondition* and *postcondition* to `0` (was `0..1`) which actually lead to the removal of both associations, constraining the *Constraint–Include* multiplicity at *Include* to `0` (was `0..1`) which again lead to the removal of the associations.

The metamodel in Figure 13 can be provided to analysts—presumably accompanied by a textual description—so they can adhere to it.

6.2 Use Case Model Interchange

The metamodel could be used to facilitate a use case model interchange between notation-specific tools based on the metamodel instances by providing a framework for an automated transformation built upon the same metamodel. In this transformation, each feature would be associated with the features that can be used as a substitute in the target notation if it does not support the original feature. For example, if absent in the target notation, alternative flows could be substituted by subflows. If the target notation does not support subflows either, main flows could be used. This rough example indicates an ordering of the substitution features would have to be defined, but in practice complex dependencies among features would have to be defined many of which could be configurable. This is one of directions for further work.

7 Related Work

Hoffman et al. [12] recently 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 by the proposed metamodel configuration, Hoffman et al. strive for a common, inconfigurable use case modeling metamodel, which is a substantial difference. Their main concern is with ensuring consistency between use case diagrams and textual descriptions. To achieve this, they include the steps in the use case flows in their metamodel. This is complementary to the metamodel proposed in this paper. However, in our metamodel, it would be necessary to support notational variants of step representation.

There have been several other attempts to propose a unified notation for use case modeling. The UML metamodel [19] is a prominent attempt of establishing a common diagrammatical use case modeling notation.

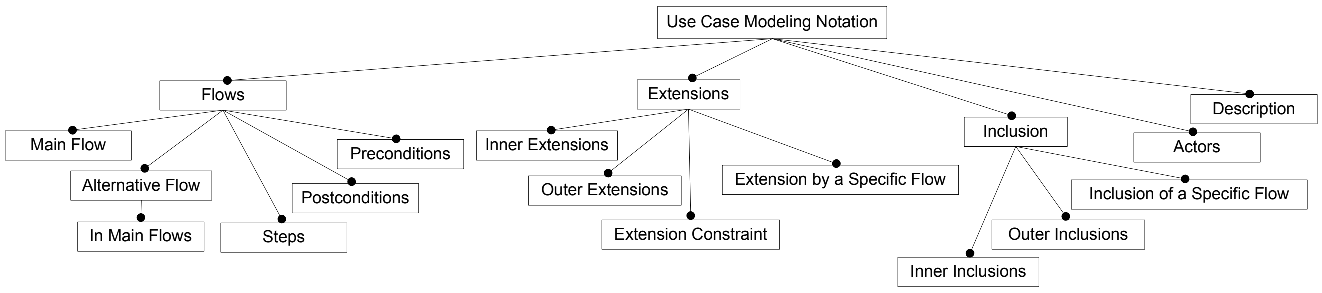


Fig. 12 Features of an example use case modeling notation.

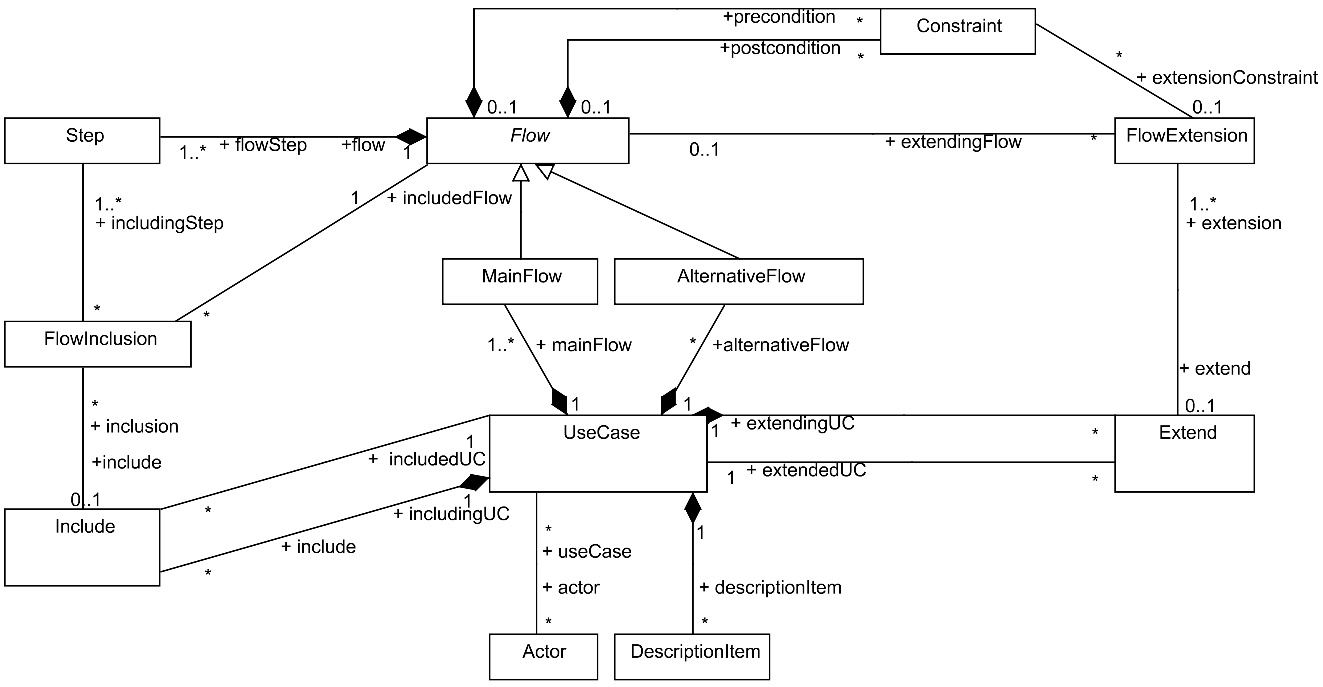


Fig. 13 A metamodel instance.

Although our aim was not to extend the UML metamodel, we used it as a basis for our use case modeling metamodel.

Rui and Butler [22] 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 [16], formalizing the include and extend relationships [4], or even formalizing use cases as such [23].

In applying the superimposed variants to the use case modeling metamodel, we relax the notion of metaexpressions proposed by Czarnecki and Antkiewicz [7] and use OCL constraints instead. Metaexpressions have originally been intended to statically determine parts of the model. Unlike metaexpressions, the OCL constraints have to remain in model variants (unless the elements they are attached to have been left out, of course). However, they can serve as a specification to rearrange the variant statically and be left out afterwards as metaexpressions are.

While feature modeling is basically in accordance with the graphical way of expressing metamodels we stick to, some constraints had to be expressed in a non-graphical form [27]. Other object-oriented metamodels have been expressed in a purely non-graphical form [21] even in cases when they define graphical models [18].

8 Conclusions and Further Work

In this paper, a configurable use case modeling metamodel with superimposed variants has been proposed to cover the existing 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, so the intention of the metamodel is not to enforce the "right" one, but to specify a family of use case modeling notation metamodels.

The use case modeling metamodel is based on the UML metamodel elements relevant to use case modeling adapted and extended to cover different use case modeling notations with a special attention paid to the elements of textual expression of flows of events in use cases.

The configuration options of the proposed use case modeling metamodel have been identified. The actual configuration option values for Jacobson's and Cockburn's notation, as the two most influential use case modeling notations, have been presented and discussed.

To better express configuration dependencies and avoid option interaction revealed in a practical evaluation by a configurable use case modeling tool prototype, the options have been arranged into a feature model and the approach of superimposed variants has been applied to the metamodel. This involved specifying presence conditions by tagged values and metaexpressions in the form of OCL expressions attached to metaclasses.

The metamodel may serve as a basis for a configurable use case modeling tool or notation-specific tools. More important, it provides a framework for a consistent application of the notation in one or across several organizations. It can also be used to facilitate a use case model interchange between notation-specific tools based on the metamodel instances by providing a framework for an automated transformation built upon the same metamodel.

We expect our metamodel would develop further to embrace other possibilities of use case modeling such as expressing the type of a use case. For example, we have not considered explicitly business use cases [11], which certainly deserve attention. We plan to extend our configurable use case modeling tool prototype and perform experiments that embrace new features.

Yet another line of further work is to improve the integration of the elements of our use case modeling metamodel with the UML metamodel. We would also like to explore the possibilities of use case model refactoring in the context of refactoring other UML models [24] and with respect to their textual representation.

Acknowledgment

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

References

1. Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange, eclipse '04*, pages 67–72, Vancouver, British Columbia, Canada, 2004. ACM Press.
2. Jim Arlow and Ila Neustadt. *UML 2 and the Unified Process.* Addison-Wesley, 2005.
3. Kurt Bittner and Ian Spence. *Use Case Modeling.* Addison-Wesley, 2002.
4. Alexandre Bragança and Ricardo 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.
5. R. J. A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, 1998.
6. Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2000.
7. Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glück and Michael R. Lowry, editors, *Proc. of Generative Programming and Component Engineering, 4th International Conference, GPCE 2005*, LNCS 3676, pages 422–437, Tallinn, Estonia, October 2005. Springer.
8. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programing: Methods, Tools, and Applications.* Addison-Wesley, 2000.
9. Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10:143–169, April/June 2005.
10. Adenekan Dedeke and Benjamin Lieberman. Qualifying use case diagram associations. *IEEE Computer*, 39(6):23–29, June 2006.
11. Jim Heumann. Introduction to business modeling using the unified modeling language (uml). developerWorks, IBM, November 2003. `http://www.ibm.com/developerworks/rational/library/360.html`.
12. Veit Hoffmann, Horst Lichter, Alexander Nyßen, and Andreas Walter. 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/article2/`.
13. Ivar Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.
14. Ivar Jacobson and Ng Pan-Wei. *Aspect-Oriented Software Development with Use Cases.* Addison-Wesley, 2004.
15. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA): A feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, November 1990.
16. Pierre Metz, John O'Brien, and Wolfgang Weber. Specifying use case interaction: Types of alternative courses. *Journal of Object-Oriented Programming*, 2(2):111–131, March 2003.
17. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.

18. Matúš Navarčik and Ivan Polášek. Object model notation. In *Proc. of 8th International Conference on Information Systems Implementation and Modelling, ISIM 2005*, Rožnov pod Radhoštěm, Czech Republic, 2005.
19. Object Management Group. OMG unified modeling language (OMG UML), superstructure, v2.1.2, November 2007. `http://www.omg.org/docs/formal/07-11-02.pdf`.
20. Tom Pender. *UML Bible.* Wiley, 2003.
21. Jaroslav Porubän and Peter 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, September 2008.
22. Kexing Rui and Gregory Butler. Refactoring use case models: The metamodel. In Michael J. Oudshoorn, editor, *Proc. of 26th Australasian Computer Science Conference, ACSC 2003*, pages 301–308, Adelaide, Australia, February 2003.
23. Perdita Stevens. On use cases and their relationships in the unified modelling language. In Heinrich 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, April 2001. Springer.
24. Miroslav Štolc and Ivan Polášek. A visual based framework for the model refactoring techniques. In *Proc. of 8th International Symposium on Applied Machine Intelligence and Informatics, SAMI 2010*, Herľany, Slovakia, January 2010. IEEE.
25. Ľuboš Zelinka and Valentino Vranić. A configurable UML based use case modeling metamodel. In *Proc. of 1st Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2009*, Novi Sad, Serbia, September 2009. IEEE Computer Society.
26. Gunnar Övergaard and Karen Palmkvist. *Use Cases: Patterns and Blueprints.* Addison-Wesley, 2004.
27. Valentino Vranić. Reconciling feature modeling: A feature modeling metamodel. In Matias Weske and Peter 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, September 2004. Springer.