Softw Syst Model (2015) 14:813–838
DOI 10.1007/s10270-013-0333-9

REGULAR PAPER

Extending the UML use case metamodel with behavioral information to facilitate model analysis and interchange

Mohammed Misbhauddin · Mohammad Alshayeb

Received: 15 December 2011 / Revised: 30 December 2012 / Accepted: 3 March 2013 / Published online: 30 March 2013

Abstract Use case diagrams are primary artifacts used for modeling functional requirements. Use case diagrams are part of the Unified Modeling Language (UML) suite of models that has become a de facto standard for modeling object oriented languages. Each model in this suite is described by a metamodel that dictates its syntax and semantics. The use case diagram is considered the most controversial diagram in UML. Practitioners claim that the use case diagram cannot be used as a valuable artifact for requirement analysis. The main reason behind this concern is the lack of behavioral description of a use case depicted within the model. Quite a few extensions to the use case metamodel have been proposed in literature to incorporate behavioral aspect of a use case within the metamodel. All these extensions omit a few important features like generalization and most of them can only be used for model representation and cannot be used for model analysis and evaluation. In this paper, we propose an extension to the UML use case metamodel with use case behavior specification elements. The main objective of the proposed extension is to provide a complete metamodel for use case diagrams which includes representation for all its elements and relationships in a conflict-free manner and one that includes information for model analysis, evaluation, and interchange among modeling tools. In order to include all valuable information related to a use case, a number of use case representation templates were considered for the proposed extension. Simultaneously, to enable the use case models generated based on the proposed metamodel to be used for analysis, pertinent information related to model usage in analysis such as effort estimation, use case scheduling, and use case metrics evaluation were considered from published studies, tools, and paradigms and included within the proposed metamodel.

Keywords UML · Use case diagram · Metamodel · Behavior specification

1 Introduction

Requirements engineering is considered as an essential front-end activity in the software development process. The majority of the effort required for requirements engineering occurs early in the lifetime of the project mainly because it is expensive to fix requirements errors such as misunderstood or omitted requirements later in the development lifecycle. Requirements elicitation is often regarded as the foremost step in the requirements engineering process. Of all the available approaches to requirements elicitation, model-driven techniques have gained immense popularity primarily because of the model-driven software development (MDSD) [1] paradigm.

Quite a few classes of approaches exist when using model-driven techniques for requirements elicitation. These include viewpoint-based approaches such as PREView [2], goal-oriented approaches such as non-functional requirements framework [3], I*[4], KAOS [5] and scenario-based approaches such as use cases [6]. The use case approach, initially introduced by Jacobson, has been integrated as part of the Unified Modeling Language (UML) [7] which is regarded

Communicated by Dr. Sebastien Gerard.

M. Misbhauddin · M. Alshayeb (✉)
Information and Computer Science Department, King Fahd University of Petroleum and Minerals, P.O. Box 5066, Dhahran 31261, Saudi Arabia
e-mail: alshayeb@kfupm.edu.sa

M. Misbhauddin
e-mail: mdmisbha@kfupm.edu.sa

as a dominant modeling language in software development [8]. Hence, the use case model is regarded as de facto standard for requirements elicitation.

A use case model represents the functional view of an object oriented (OO) system and plays a vital role in modeling the system's functional requirements. The use case model represents the functional requirements as a set of use cases. Each use case is a specification of a set of operations between the system and the actors resulting in an output valuable to actors or stakeholders of the system. UML use case diagram models use cases and their relationships with actors and other use cases. Behavior of each use case is typically documented either through other UML models (sequence [9–11] or activity diagrams [12–14]), formal modeling languages [15–18], or as natural language text. Most researchers and practitioners agree that use case behavior written in the vocabulary of the problem domain is better understood by non-technical stakeholders than any other notation. Hence text-based approaches gained immense popularity. Two major advantages available by selecting the text-based approach for behavioral specification are (1) understandable by both technical and non-technical stakeholders and (2) minimum use of UML vocabulary.

Taking advantage of the model-driven development initiative, UML allows OO practitioners to extend or even modify the base language metamodel to adapt the language to a specific situation or domain. A metamodel is a model of a modeling language and metamodeling is the process of developing a metamodel that specifies concepts and their relationships for the purpose of interpreting models. Utilizing the metamodeling approach, a number of proposals for integration of use case behavior within UML-specified structural use case model have been proposed. Most of these approaches either lack complete behavior specification, use complex formal notations that creates communication gap, or lack vital information for use case analysis and evaluation.

Extensions to the use case metamodel proposed by [19–22] fail to include the concept of flows (or scenarios) within use case descriptions. Extensions proposed by [23–26] explicitly modeled flows as a set of steps within a use case description. Although these works modeled use case flow, the lowest level of abstraction in their work is a use case step of which a flow is composed of. Almost all extensions proposed to the UML use case metamodel do not model the generalization relationship except for the metamodel proposed by Repond et al. [27]. Although the metamodel extension proposed by Repond et al. models the generalization relationship, it has two major constraints: specialized use case can only add additional behavior and allows the generalized use case to have knowledge of which use cases specialize it and where they add additional behavior.

In order to allow the specification of use case behavior in an unconstrained manner that can be easily comprehended by non-technical stakeholders and providing a formal structure for use case analysis at the same time, we propose an extension to the UML use case metamodel. The main reason for proposing an extension to the use case metamodel is that the existing extensions are either incomplete (i.e. lack specification of important features such as generalization) or do not model information pertinent to use case analysis. Text-based descriptions of use case behavior usually follow a use case template. Since there is no standardized use case template, we base our metamodel extension on a custom template developed after in-depth assessment of a number of available templates. One of the main purposes of a metamodel is to provide a schema for exchanging semantic data. Since eXtensible Markup Language (XML) has become a text format widely used in the exchange of varied data on the web and elsewhere, especially for porting UML models, we provide an XML Schema [28] for the extended metamodel to facilitate use case model storage, exchange, transformation, analysis, and evaluation. An XML Schema dictates the structure of an XML document to ensure its well-formedness.

The paper is further organized as follows: in the next section, we provide background information on the UML use case diagram, its metamodel and a summary of different approaches available for describing use case behavior. In Sect. 3, we describe a complete step-by-step extension of the use case metamodel. Section 4 describes our implementation of the extended metamodel as part of a use case modeling tool, UCDesc. Section 5 illustrates usability of the metamodel through its various applications for use case analysis. Related Works are discussed in Sect. 6 followed by a list of validity threats for the proposed extension to the use case metamodel in Sect. 7. Finally, Sect. 8 concludes our work and provides directions for future work.

2 Background

2.1 UML use case model

A use case diagram consists of four distinct elements that depict the working of a system: The system itself, the actors that interact with the system, the services (or use cases) the system is required to perform, and the relationships between these elements. The system element sets the boundary of the system with respect to the actors who use it and the services it must provide. Actors are depicted outside the system element boundary as they are not realized by the system and services are depicted inside the system element. The notion of a system element is to establish the scope of the system.

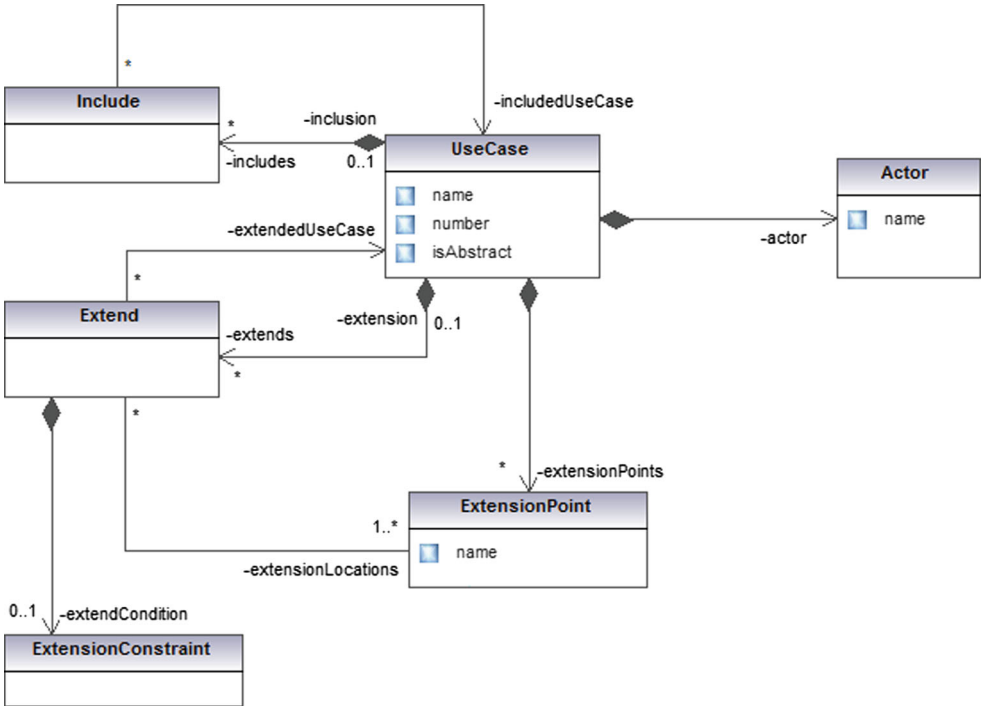


Fig. 1 A subset of the UML metamodel [7] for use case diagram

An actor in a use case diagram can be associated with one or more use cases. This relationship can specify whether the actor initiates the use case or receives results from the use case or both. An actor–use case relationship is also known as association. Although not explicitly mentioned in the UML Specification, UML provides one actor–actor relationship called generalization. UML allows three different relationships between use cases: Generalization, Inclusion and Extension. Use case generalization is similar in definition to actor generalization where general functionality is separated from specific functionality in different use cases. Two use cases are related by inclusion if one use case uses the functionality offered by the other use case. An extension relationship exists between a two use cases when one use case wants to utilize the functionality of another use case if certain conditions are satisfied.

UML models are described by a metamodel detailed out in its specification document [7]. A UML metamodel is a qualified alternate of the UML models and is a representative of any model that can be expressed with it. Since the UML metamodel includes information for all the models in the modeling suite, a subset of the UML metamodel that includes all elements related to modeling a use case diagram is shown in Fig. 1.

2.2 Use case description

The use case model that is part of the UML specification describes only its structural view. The structural view defines the services provided by the system without divulging its internal structure. The internal structure presents the behavioral aspect of the use case. A use case, once initiated by an actor, performs a number of operations to provide a meaningful output to the invoking actor. These set of operations constitutes a use case's behavior. There are a number of ways in which the behavioral information can be presented. A classification of these approaches is given in Fig. 2 below:

Of all the available approaches, text-based approaches gained immense popularity as they allow for high level of stakeholder comprehension and involvement which is the main goal for use case specification. One major trade-off when selecting textual specifications to model use case behavior is that they are prone to mistakes and incompleteness. Although using formal models and other UML diagrammatic notations for requirements elicitation and use case description allows for better structure and validation, it also introduces a high participation hurdle for customer involvement which is the main goal for use case specification. In general, a typical structure of a use case behavior is illustrated in Fig. 3.

The use case behavior consists of two major parts of information: description (description part) and dynamics (consisting of flows). Text-based approaches usually follow a template that describes these information elements in a structured fashion. Table 1 shows a number of prevalent initiatives that describe use case behavior in the form of a structured template.

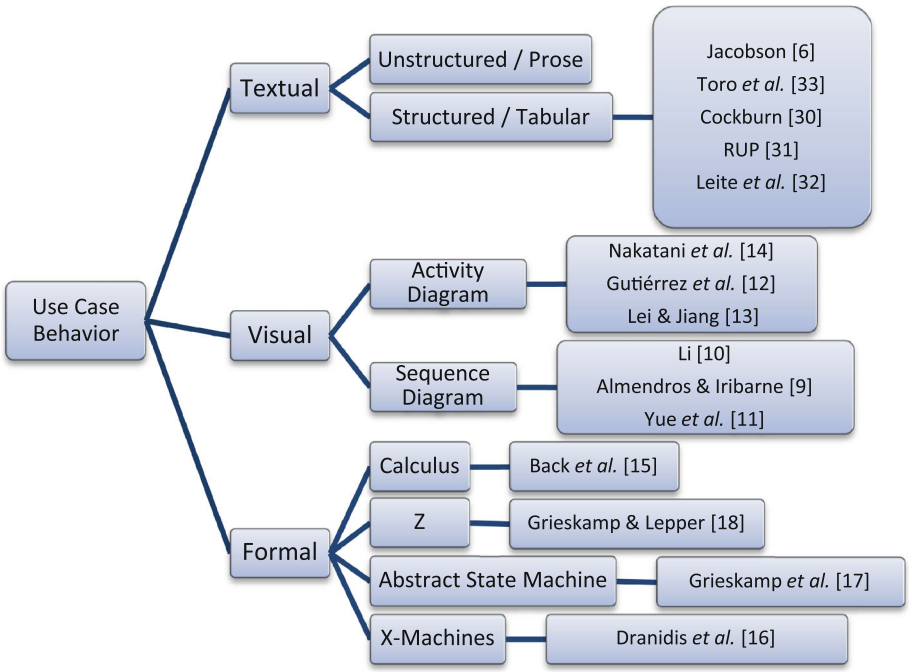


Fig. 2 Use case behavior description approaches

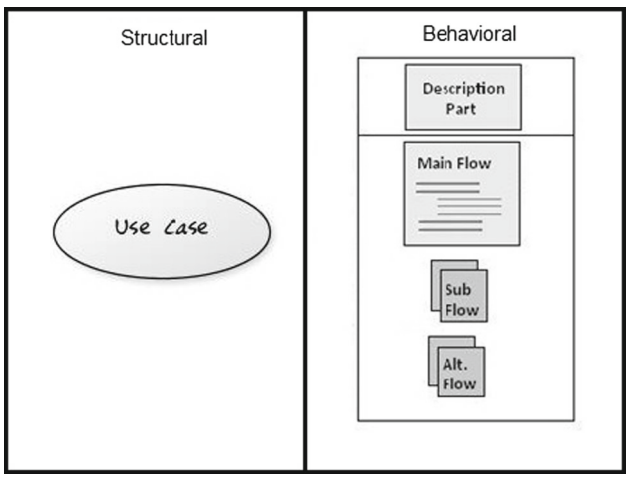


Fig. 3 Structure of a typical text-based use case description

3 Use case metamodel extension

UML provides two mechanisms for extending its base metamodel: Lightweight and Heavyweight extensions. Lightweight extensions do not add new model elements to the existing UML metamodel. UML profiles are used to implement these types of extensions. A UML profile is a predefined set of stereotypes, tagged values, and constraints to support modeling in specific domains. Profiles give a well-defined manner of adopting the standard UML model to a particular domain. Since a profile is not a new element, its expressiveness is constrained by the model element it specializes. Adding new elements in the form of meta-classes, defining suitable metadata and meta-associations is referred to as heavyweight extension. Heavyweight extensions are guided by the meta-object facility's (MOF) meta-metamodel language [29]. In this paper, we propose a heavyweight extension to the UML metamodel for use case diagram mainly because this extension mechanism allows addition of any desired feature to the metamodel without constraining its expressiveness.

In this section, we describe the extension to the use case metamodel of UML by including elements necessary to describe its behavior. For the sake of clarity of presentation, we construct the metamodel in pieces. A complete metamodel will be presented towards the end of this section. As observed from Table 1, a number of variations exist in the elements for use case description template. Despite these differences, each approach has two major parts of information: description and dynamics separated by a thick line shown in Table 1. The description part includes elements such as name, number, goal, scope, level, description, actors (primary and secondary), preconditions, post-conditions (success and failure), priority, frequency, open issues, due date, and special requirements. The dynamics part captures the use case's flow of execution. Flow of execution of a use case includes a sequence of steps that can either be events (messages exchanged between actors and use case objects), or anchors (that disrupt the main flow by allowing access to sub flows, alternate flows, use case extensions and inclusions).

The main objectives in proposing an extension to the use case metamodel can be summarized as follows:

1. The original metamodel is an essential subset of the extended metamodel so that information can be utilized

Table 1 Template elements from different notation proposed in literature

Template elements	Cockburn [30]	Jacobson [6]	RUP [31]	Leite [32]	Toro [33]
Name: unique name assigned to a use case	√	√	√	√	√
Number: unique ID assigned to a use case	√			√	
Goal: statement of goals expected from the use case	√				
Scope: system being considered black-box under design	√				
Level: level of use case description	√				
Description: brief summary of use case purpose		√	√		
Primary actor: actor that initiates the use case	√				√
Secondary/supporting actors: actors that participate within the use case	√				
Offstage actors: non-interacting actors concerned with the outcome of the use case					
Special requirements: list of non-functional requirements		√	√		
Preconditions: expected state of the system prior to use case execution	√	√	√	√	√
Post-conditions (Success): state of the system upon successful completion of the use case	√	√	√		√
Post-conditions (failure) state of the system if goal is abandoned	√				
Performance target: the amount of time this use case should take	√				
Priority: how critical to the system/organization is the use case	√				
Frequency: how often is it expected to happen	√				
Open issues: list of issues about this use case awaiting decisions	√				√
Due date: date or release of deployment	√				
Main flow: Steps of the scenario from trigger to goal delivery	√	√	√	√	√
Sub flows: Sub-variations that will cause eventual bifurcation in the flow	√	√		√	
Alternate flows: conditional variations that will cause eventual bifurcation in the flow	√	√		√	
Extension points: list of extensions each referring to a step in the main flow		√	√		
Exceptions: conditional variations that will cause unsuccessful termination of use case flow				√	√
Super use case: name of use case that this one specializes	√				
Sub use case: links to all use cases that specialize this use case	√				

from both depending upon the requirement of the user. In order to ensure a clean extension, meta-classes re-used from the UML specification are grayed-out, re-used meta-relationships are bolded-out, and meta-attributes added to re-used meta-classes have a distinct shape added to the symbol.

2. The extended metamodel should take into consideration information from all published templates. But information that is useful for further analysis of the use case model should be included as meta-classes so other tools can access and extend it easily and other information can be included as meta-attributes of the respective meta-classes.
3. Information for use case analysis, model evaluation, and model interchange should be readily available and accessible from the metamodel.
4. The extended metamodel should provide an integrated global modeling environment for tools and users and provide seamless transition from requirements to system modeling.

3.1 Actors

Actors are used in the use case diagram to model users of the system. The UML Specification defines actors as entities that can communicate with several use cases. In this proposed extension to the use case metamodel, we classify actors based on two criterions: the role they play in a use case and the role they play in the system. Many authors define different types of actors based on their role in the use case. According to Larman [34], an actor can be classified into three types:

1. Primary actor: an actor that initiates the use case and helps realize its goal.
2. Supporting actor: an actor that participates in a use case that helps realize a primary actor's goal.
3. Offstage actor: an actor that does not interact with the system but has needs that should be addressed in the system. Offstage actors are considered as stakeholders of the system under development.

The actor's type may differ from one use case to another. Based on the above classification, we added three associations between the *UseCase* meta-class and the *Actor* meta-class to denote the role an actor plays in a use case. Popularity of the use of use case modeling as a de facto standard for requirement modeling in the field of software engineering was further enhanced with the establishment of a software estimation technique known as use case points (UCP) [35]. UCP became a good candidate for early estimation of software size and effort because of its simplicity and ease of use. The main activity of UCP is to estimate the complexity of actors and use cases. The complexity of actors is identified based on the role an actor plays in the system (as opposed to in a use case as discussed above).

In order to incorporate this information in our proposed metamodel, we categorized actors based on information from both the original UCP model presented by Schneider and Winters [35] and the enhanced model known as iUCP presented by Nunes [36]. Based on this, we classified the actors into the following categories:

- System actor: this type of actor is another system interacting with the base system through an application programming interface (API). For example, the ATM system reads the credit card information directly from a credit card reader. In this case, the credit card reader is outside the system and accessed through an API. Hence, the credit card reader is a system actor.
- Network system actor: this type of actor is another system interacting with the base system through a protocol or data store. For example, the ATM system verifies the credit card information from an accounting system. In this case, the accounting system is outside the system and is accessed through a network. Therefore, the accounting system is a network system actor.
- Human actor: this type of actor is a person or a user who will use the system. It is the most common type of actor. For example in the ATM system, a customer will ask the system to perform a transaction and, therefore, the customer is a human actor.

The iUCP model differs from the original UCP model as it is based on the usage-centered design method [37] in contrast to the conventional use case model for classifying actors. The main reason behind this is the richness of the information conveyed by the usage-centered method regarding the complexity underlying each actor. Human actors are divided into simple, average, and complex, based on the number of roles they play in the system. In the usage-centered design method, the concept of actor is expanded through user roles that represent the relationship between users and a system. A user role is characterized by the "*context in which it is performed, the characteristic manner in which it is performed, and the design criteria for the role's supporting performance*" [36]. The number of roles supported by each human actor provides an important way to infer the complexity associated with each actor. In order to incorporate this, we added a meta-attribute called *num_roles* to the *Actor* meta-class. Since this attribute is associated with human actors only, a default value of 1 is used for system and network system actors.

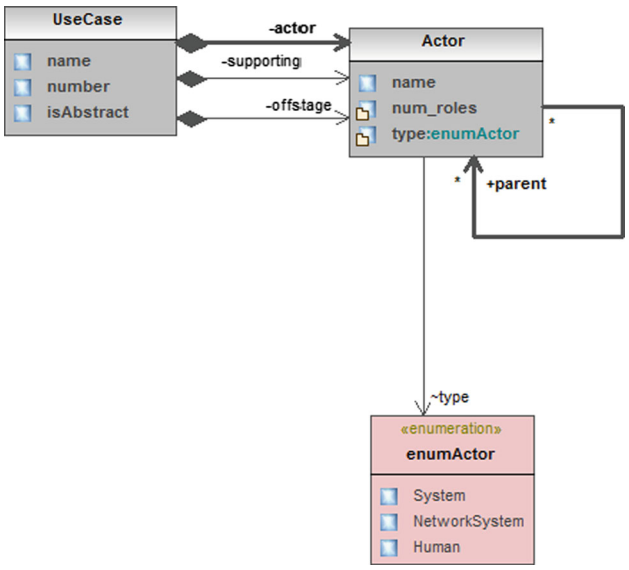


Fig. 4 Addition to the extended UML metamodel for actor

Actors in a use case model can be associated with each other using the generalization relationship. It is the only kind of relationship that exists between actors. The actor modeling the common role is referred to as the parent actor and the actors using the common role are called the child actors. In simple terms, a child actor inherits the capability to communicate with the use cases its parent actor is associated with. The metamodel representation with the modified *Actor* meta-class and its relationship with the *UseCase* meta-class are presented in Fig. 4.

3.2 Use case

A use case within a use case model consists of numerous information elements shown in Table 1. Despite the difference of information portrayed by different templates, each template has two major parts of information: the description part and the dynamics part. The description part includes elements such as name, actor, preconditions and so on. The dynamics part captures the use case's flow of execution. Flow of execution of a use case includes a sequence of steps that can either be or anchors.

Information within the use case description can be classified into two categories: information that is used for "mere" documentation purpose and information that will

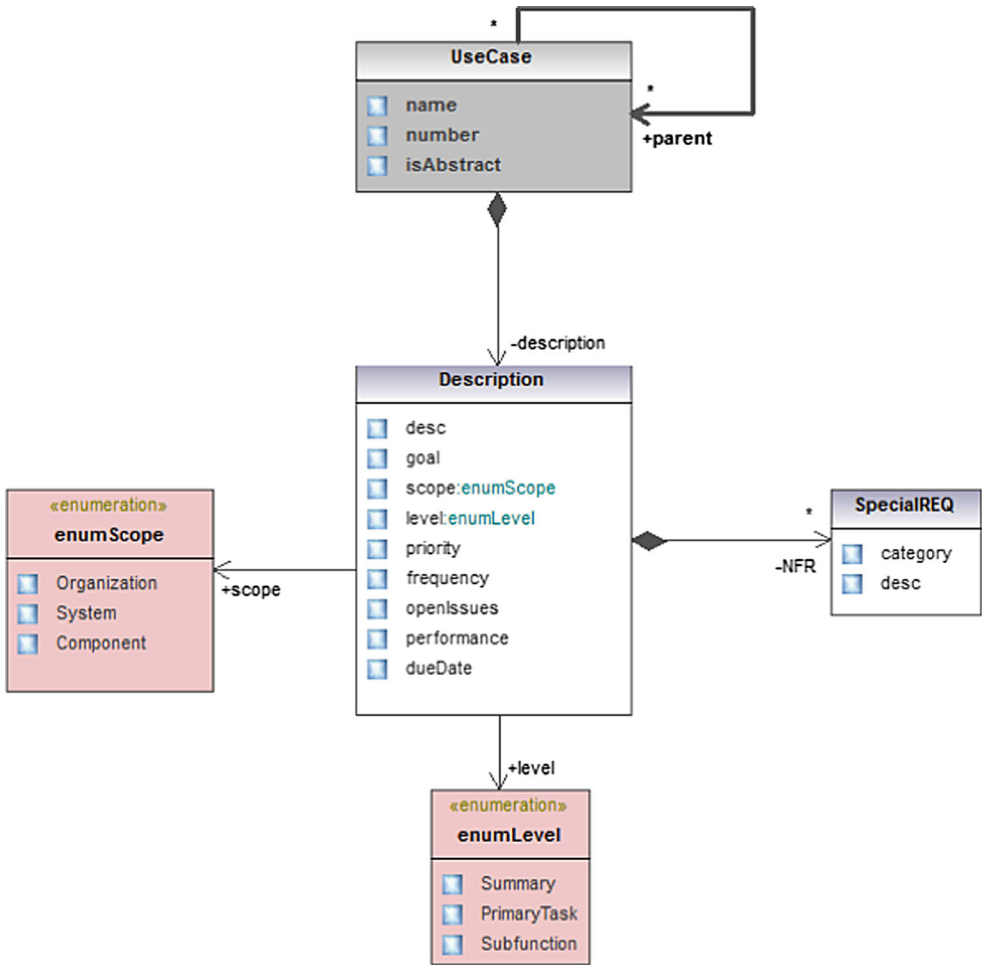


Fig. 5 Addition to the extended UML metamodel for use case

be used for use case analysis at later stages of software development. Keeping in line with the above-mentioned criteria, we decided to separate these elements and depict them independently in the enhanced metamodel as follows:

1. Use case description elements that will be used for its documentation will be represented as meta-attributes in a separate meta-class called *Description* (See Fig. 5).
2. Use case description elements that will be used for analysis will be represented as separate meta-classes and elaborated and justified later throughout Sect. 3.

Cockburn [30] dedicated a section in his use case template for special requirements. This section is used to document non-functional requirements not covered by the use case dynamics but will influence the design model. Due to the multiplicity of this description, we added it to the proposed metamodel as meta-class *SpecialREQ*. Special requirements can be organized in categories such as usability, reliability, performance, and so on.

The two main components that are included in the dynamic part of the use case specification are the use case scenario description and constraints (pre and post conditions). There are many forms of use case specifications formats available in the literature. Representing a use case scenario as a collection of numbered sentences is the current best practice used in documenting use cases [6,30]. A use case scenario is expressed through a sequence of steps. These steps are grouped to form behavioral fragments called flows. The extension of our use case metamodel to incorporate different flows (main, sub and alternate) within a use case behavior is discussed in Sect. 3.4. In Sect. 3.5, we show the extension of use case model to express the various constraints.

3.3 Use case relationships

UML defines three types of relationships between use cases: «include», «extends», and generalization. When describing these relationships through a metamodel, we need to discuss the relationship depiction on the use case structural view

and within the use case flow of execution (its behavioral view). In this section, we discuss the impact of use case relationships on metamodel elements that depict the use case's structural view. We provide a coherent description of these relationships derived from the literature and extend the use case metamodel based on these descriptions. The manner in which these relationships are depicted in the use case's flow of execution is discussed in Sect. 3.4.

3.3.1 Include relationship

Two use cases are related by the «include» relationship if one use case (known as the base use case) uses the functionality offered by the other use case (known as the included use case). Two main reasons for using the «include» relationship in a use case model according to the UML specification are to fragment Complex Use Case into manageable ones [34,35] and to reuse use Cases [34,35,38–42]. Apart from this, some authors recommend the use of «include» relationship for conditional behavior [34,39,41] and to handle asynchronous events [34]. The main motivation behind the use of «include» relationship for conditional behavior by the above-mentioned authors is that this relationship is much easier for most people to understand and use than other relationships such as «extends» and generalization. Also the use of «extends» is restricted to cases where the base use case is locked or "closed for modification". Since it is difficult to gauge when a use case is closed for modification, we adopted the semantics of the «include» relationship as outlined in the UML specification and accepted by majority of the authors [43] and leave the concept of conditional behavior to the «extends» relationship. We do not modify the meta-classes related to the «include» relationship in the extended metamodel.

3.3.2 Extend relationship

Two use cases are related by the «extends» relationship if one use case (known as the base use case) implicitly incorporates the behavior of another use case (known as extension use case) at a specified location. The extension use case is executed only when some particular condition is satisfied in the base use case. There have been many reasons proposed in literature for the use of the «extends» relationship in the use case model. These can be summarized as follows:

1. Optional or exceptional behavior: behavior that is optional to the base use case can be separated and defined in an extending use case. Most authors agree with this usage of the «extends» relationship [35,38–41,44].
2. Asynchronous events: an asynchronous event is one that can be called at any point in the base use case. The use of the «extends» relationship to describe asynchronous events is supported by Constantine and Lockwood [44] and Cockburn [30].
3. Defer behavior implementation: Armour and Miller [39] suggested the use of «extends» relationship to separate behavior from the base use case that can be developed later to assign it a lower priority.

The semantics of the «extends» relationship has created a lot of disagreement among authors. In this section, we attempt to resolve these concerns by extending the metamodel to incorporate necessary information to ensure consistency in semantics of this relationship. Since the «extends» relationship is optional and controlled by an execution condition, it requires the specification of the following elements:

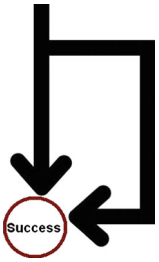
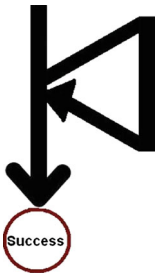
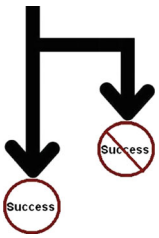
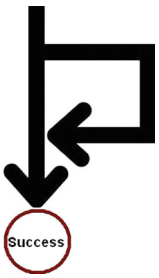
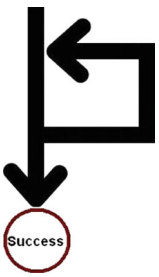
- Extension point: the point in the behavior of the base use case where an extended use case can be inserted is known as the extension point.
- Extension constraint: this is an optional constraint that specifies the condition that must be true for the extension use case to be invoked from the base use case.

When the extension point in base use case scenario is reached, the extension constraint is evaluated and the control is switched to the extension use case. After the execution of the extension use case, the control is resumed just after the extension point in the base use case scenario [6,30,39]. But in order to use the «extends» relationship to model exceptional behavior, the control should be allowed to return to any point in the base use case flow or be allowed to end the use case resulting in a failure or alternative success scenario. In order to handle these situations, Metz et al. [21] defined five types of alternative sequences. These are summarized in Table 2.

In order to accommodate the sequences mentioned in Table 2, the concept of rejoin point was proposed [19,21]. A rejoin point allows the control to return to separate point in the main flow after performing the steps specified in the extension use case. We followed a similar approach in our proposed extension of the use case metamodel and added a meta-class called RejoinPoint. When the rejoin point is equal to the extension point it leads to an alternative insertion fragment. When the rejoin point is a point that occurs either before or after the extension point, then the alternate scenario leads to an alternative cycle or alternative fragment respectively. Finally, when the rejoin point is not specified, it leads to a use case exception.

In order to complete our extension to the use case metamodel for «extends» relationship, we considered an interesting premise put forward by Laguna and Marqués [45]. An extension point in the base use case can be extended by several use cases. An issue arises when this extension point is reached and a decision is to be made whether only

Table 2 Summary of alternative scenarios adapted from [21]

Alternative history The control in this type of alternative sequence never returns back to the base use case scenario. The success post-condition in this case can either be the overall success post condition of the base use case or its subset	
Alternative insertion The control in this type of alternative sequence returns to the point just after the extension point in the base use case	
Use case exception The control in this type of alternative sequence never returns back to the base use case scenario. In contrast to alternative history, the use case exception is always a failure scenario and results in a failure post condition	
Alternative fragment The control in this type of alternative sequence returns to any point after the extension point in the base use case	
Alternative cycle The control in this type of alternative sequence returns to any point before the extension point in the base use case	

one or at least one among these extension use cases is to be selected. In order to complete and clarify the behavior of the base use case and to aid in the process of elicitation of requirements, Laguna and Marqués [45] added multiplicity attributes to the extension point meta-class. Following their approach, we added the *lower* and *upper* meta-attributes to the *ExtensionPoint* meta-class to clarify the behavior of «extends» relationship in case of multiple use case extensions. A multiplicity of 0..1 states that the extension use case can be executed when the constraint is true (equivalent to the original UML «extends» semantics), a multiplicity of 1..1 states that only one of the possible extension use case can be selected, and finally a multiplicity of 1..* allows more than one use case to be inserted.

In addition, following Constantine and Lockwood [44] in our proposed metamodel extension, we considered the concept of asynchronous extensions in which an extension use case can be called asynchronously at any step of the use case flow. Asynchronous extensions are defined in our metamodel as a separate meta-class called *AsyncExtend*. It is defined separately as it lacks an extension point and extension location. For example, a customer can press cancel at any time during his usage of the ATM Machine. Figure 6 shows the extended metamodel for «extends» relationship.

3.3.3 Generalization relationship

The generalization relationship in a use case model allows a given use case to be defined as a specialized form of an existing use case. Common behaviors, constraints and assumptions are factored out into a general use case (also known as the parent use case) which can then be inherited by a specialized use case (also known as the child use case). The concept of generalization and specialization gives rise to two types of use cases: Abstract and Concrete. An abstract use case is an incomplete use case which can only be invoked by another use case. On the other hand, a concrete use case is a self-contained complete use case that can be directly invoked by an actor.

Most researchers agree with the definition and use of the generalization relationship. Figure 5 depicts the use case metamodel for generalization. Although the structural representation of this relationship is fairly straightforward, its usage within a use case scenario description is vaguely described in literature. A metamodel for generalization within a use case description is discussed in Sect. 3.4.

3.4 Use case flows

From the many forms of composing the dynamics part of the use case specification, Bittner and Spence [41] provided the most promising one. They expressed the use case dynamics through a sequence of steps. These steps are grouped to form behavioral fragments called flows. A single use case consists of multiple flows as shown in Fig. 3, but the flow of events that is initiated when the use case is executed by an actor is called the main flow. Apart from the main flow, a use case can

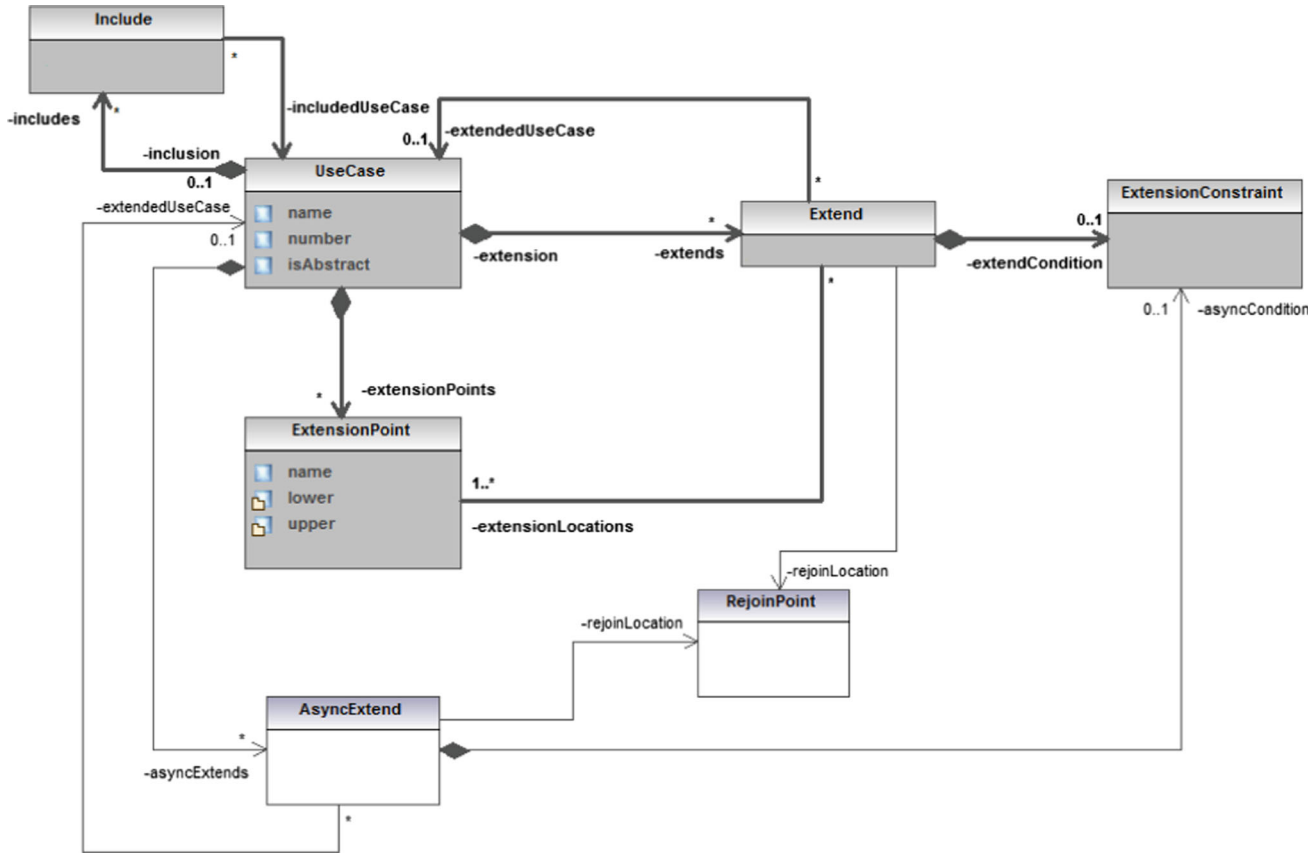


Fig. 6 Addition to the extended UML metamodel for «extends» relationship

also have multiple sub flows and alternate flows. These flows are initiated from the main flow. A sub flow is used either to describe complex logic associated with a particular step or to factor out redundant steps described in a flow. Alternate flows include behavior that is alternate to the use case. This could be optional or exceptional behavior steps within a flow, the content interpretation of which will be discussed later in Sect. 3.4.2. Usually, unconstrained natural language is used to describe the steps within a flow.

Following the flow composition architecture described in Fig. 3, we initially added a meta-class called *Flow* to the extended use case metamodel. Different types of flows are then represented as specialized meta-classes of the *Flow* meta-class: *MainFlow, SubFlow* and *AlternateFlow*. Apart from terminological differences and elements used, there are some noteworthy semantic differences between the templates mentioned earlier in Table 1. In order to ensure deterministic initiation of use cases and their completeness, we describe the semantics that our extended metamodel is built upon as follows:

1. Restrict the number of main flows to only one (as described by Cockburn [30] and opposed to Jacobson's notation [6] that allows multiple main flows).
2. Allow sub flows and alternate flows within sub flows and alternate flows.
3. Allow multiple extension points (as described by Jacobson's notation [6] and opposed to Cockburn's notation [30] that does not allow flow extensions at all).

In order to allow sub flows and alternate flows to have sub flows and alternate flows within them, we added another level of inheritance between the *Flow* meta-class and *SubFlow* and *AlternateFlow*. This intermediate meta-class is called *OtherFlow*. Most researchers define use case flow as a composition of a sequence of steps [25,27,30]. Since one of our main goals for extending the use case metamodel is to use the instantiated use case model for analysis, we used the concept of transactions. One main reason of using transactions to describe flows is that transactions are mainly used as a complexity metric within the UCP method. A transaction is a shortest sequence of use case steps starting from an actor's request and ending in a system response [46]. Hence a use case flow is composed of a number of ordered transactions included in the metamodel by the *Transaction* meta-class. Each transaction is then composed of a sequence of steps modeled by the *Step* meta-class. Figure 7 shows the excerpt of the extended metamodel for the use case flow of events.

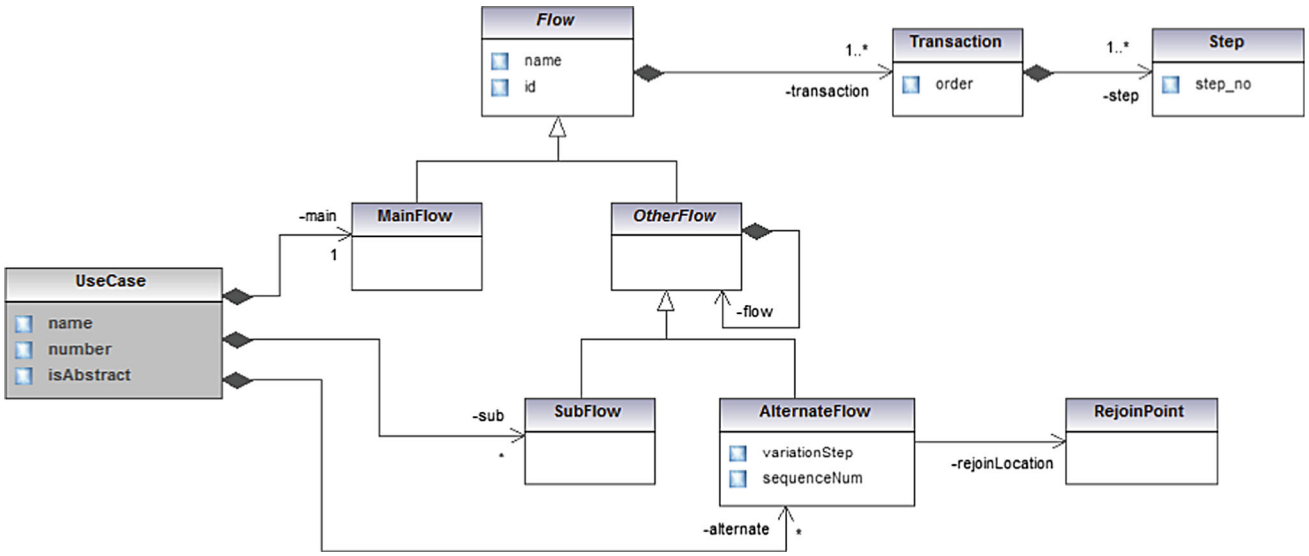


Fig. 7 Excerpt of the extended metamodel for the use case flow of events

3.4.1 Use case action steps

In a flow description, a step can be classified either as an event step or a branching step. A step that performs a certain action (from the actor to the system or vice versa) is referred to as an event. A branching step is a step that alters the sequential order of the flow by invoking the behavior of another flow of events. Branching steps are discussed in the next subsection. Natural language sentences are used to describe an event. A number of approaches that make use of the grammatical structure of the natural language and natural language processing (NLP) techniques, to analyze and extract relevant information, have been proposed in literature [47–55]. As far as the metamodel is concerned, we focused on the elements that make up a typical event sentence. An event allows a sender to communicate with one or more receivers through a message (action) that may or may not include additional parameters (arguments). Hence it is safe to assume that an event is composed of a sender, multiple receivers, an action and zero or more arguments.

Since a step can either be an event or a branching action, it is specialized by two meta-classes called *Event* and *Anchor*. The *Event* meta-class is further extended to include *Sender, Receiver, Action*, and *Argument* meta-classes based on the above-mentioned reasons. In addition, following Diev's transaction definition [46] and the transaction model proposed by Ochodek and Nawrocki [56], we enumerated four types of actions relevant from the use case transaction point of view. This is shown through an enumerated meta-attribute called *action Type* in the *Action* meta-class. Excerpt of the metamodel depicting the meta-classes relevant to a use case step is shown in Fig. 8.

3.4.2 Use case branching steps

As mentioned earlier, a step can either be an event or a branching action. We refer to the branching action step as anchors as they are mere placeholders or locations within the main flow that invoke another flow or use case. The natural order in which steps occur within a flow is sequential from top to bottom. This concept of sequential ordering can be altered by including the behavior of another flow in the main flow. A flow may include another flow in its execution. This insertion can either be conditional or unconditional. Unconditional insertions of a flow are referred to as Inclusion. A flow may include another flow which is part of the same use case description (also known as subflows) or may include a flow defined in another use case description (i.e. use cases related to each other by the UML include relationship). These two inclusions are referred to as Internal Inclusion and External Inclusion, respectively. An internal inclusion anchor specifies the name of a sub-flow (bolded out to differentiate) [6], whereas an external inclusion anchor is composed of the keyword *include* followed by the name of the use case to be included [6,30].

Use case descriptions, apart from allowing unconditional insertions, also provide a means of including another flow based on a condition. Conditional insertions of a flow are referred to as a Variation. Similar to that of Inclusion, a flow may include a variation flow part of the same use case description (also known as alternate flows) or may include a flow defined in another use case description (i.e. use cases related to each other by the UML «extends» relationship). These two variations are referred to as Internal Variation and External Variation, respectively.

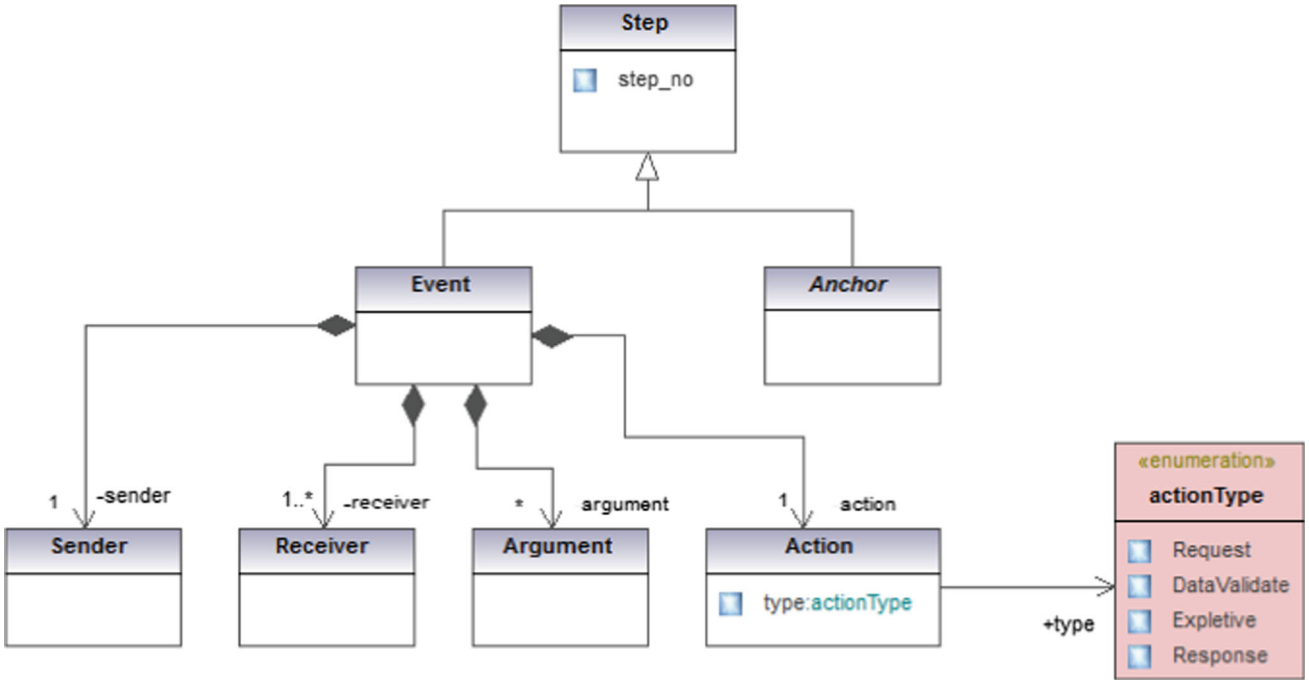


Fig. 8 Excerpt of the extended metamodel for the use case flow steps

Internal Variation anchors usually do not include branching information. Information about an alternative flow is specified in the alternative flow itself. An example of an internal variation scenario is shown in Fig. 9. Based on the example illustrated in Fig. 9, we modified the *Alternative-Flow* meta-class shown in Fig. 7 with the following meta-attributes: *variationStep* and *sequenceNum* (for cases when a single step in the main flow can result in multiple alternative flows). Since the internal variation is a conditional branch, a constraint element needs to be added to the extended meta-model. All discussions related to constraints are deferred to Sect. 3.5. Also since the alternation scenarios depicted in Table 2 are applicable to alternative flows, an association is added between the AlternativeFlow meta-class and the RejoinPoint meta-class.

An external variation anchor specifies the name of the extension point. The extension point includes information regarding the condition, location and which use case to invoke. An example of the use of an extension point and its description is shown in Fig. 10.

Figure 11 illustrates how the concepts mentioned above can be included as specialized metaclasses of the *Anchor* metaclass mentioned in Fig. 8.

Main Flow

1. ----------
2. ----------
3. The Customer enters the withdrawal amount.
4. ----------

Alternative Flow

3 (a) ATM System has no currency

1. The system notifies the customer that the ATM is out of cash.
2. The use case ends.

Fig. 9 Use case description example depicting the use of alternative flow

3.4.3 Use case generalization

One area when describing textual use case metamodels that has been given least attention is how a specialized flow of a child use case is specified. Hoffman et al. [24] were the first to discuss generalization within use case flow. They introduced the concepts called general narrative description and specialized narrative description to differentiate between original use case flow and inherited use case flow. Although the formalization provided by them has its own merits, inheriting all elements of the general narrative description within the specialized description causes redundancy and makes the behavioral model difficult to maintain. The only other work to discuss generalization in use case flow was carried out by Repond et al. [27]. In their work, a generalized use case is

Main Flow

1. ----------
2. System requests withdrawal amount
3. The Customers ent er the withdrawal amount
4. {Exceeds Limit}
5. ----------

Extension Points

Exceeds Limit [amount>=max Amount] : *Constraint*

Exceed Error , *Extension Use Case*

return:2 *Rejoin Point*

Fig. 10 Use case description example depicting the use and structure of extension points

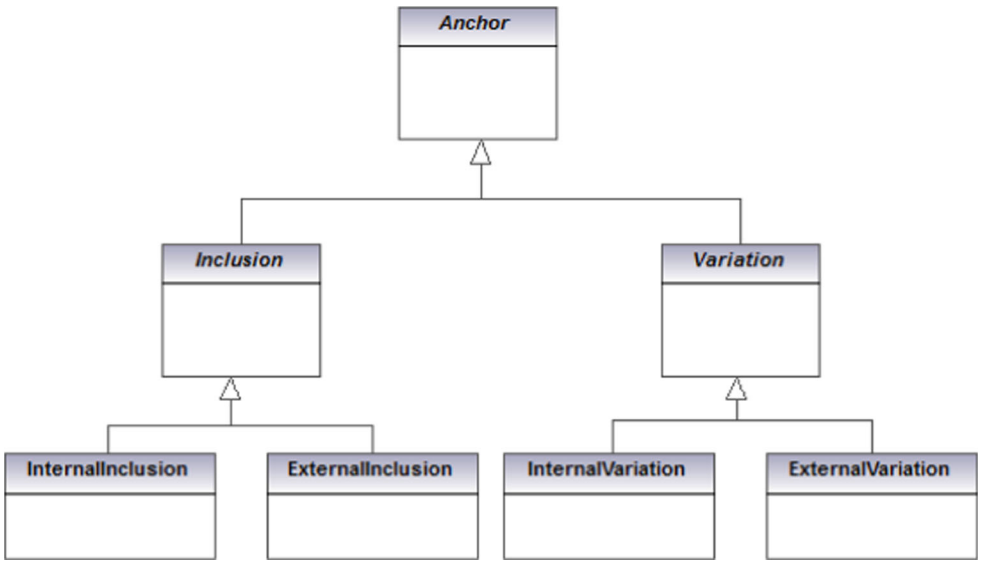


Fig. 11 Metamodel for the anchor metaclass mentioned in Fig. 8

required to define points (called Generalization Points) where the specialized use cases can add additional behavior. Two main problems with their approach are as follows:

- Specialized use case can only add additional behavior but cannot modify or replace the steps of the generalized use case.
- The use of "Generalization Point" within the generalized use case defeats the purpose of allowing the generalized use case not to care about what specialization use cases exist.

In this section, we clarify the semantics of use case generalization and provide an extension to the use case metamodel. We used the terms parent use case to refer to the generalized use case and child use case to refer to the specialized use case. The two main functions of the child use case when inheriting from a parent use case are the following: modifying existing behavior and adding new behavior. The child use case replaces a portion of actions, conditions, and rules of the parent use case. The steps to be replaced are rewritten and steps not rewritten are executed as in parent use case. Apart from this, new actions, conditions, and rules can be added, thus enhancing the behavior of the child use case. Since the flow description of a child use case will be either adding new behavior or inheriting existing behavior from the parent use case, we included it as a separate meta-class called *ChildFlow* inheriting from the *Flow* meta-class. Since the use case can either have a *MainFlow* or a *ChildFlow* depending on whether it is a parent use case or child use case, we modified the multiplicities on these two associations in the metamodel to 0..1 instead of 1.

Steps in the child use case flow can be defined locally (added behavior) which is handled by association between the super meta-class *Flow* and *Transaction* in the metamodel. Inherited behavior can either be modified or executed and

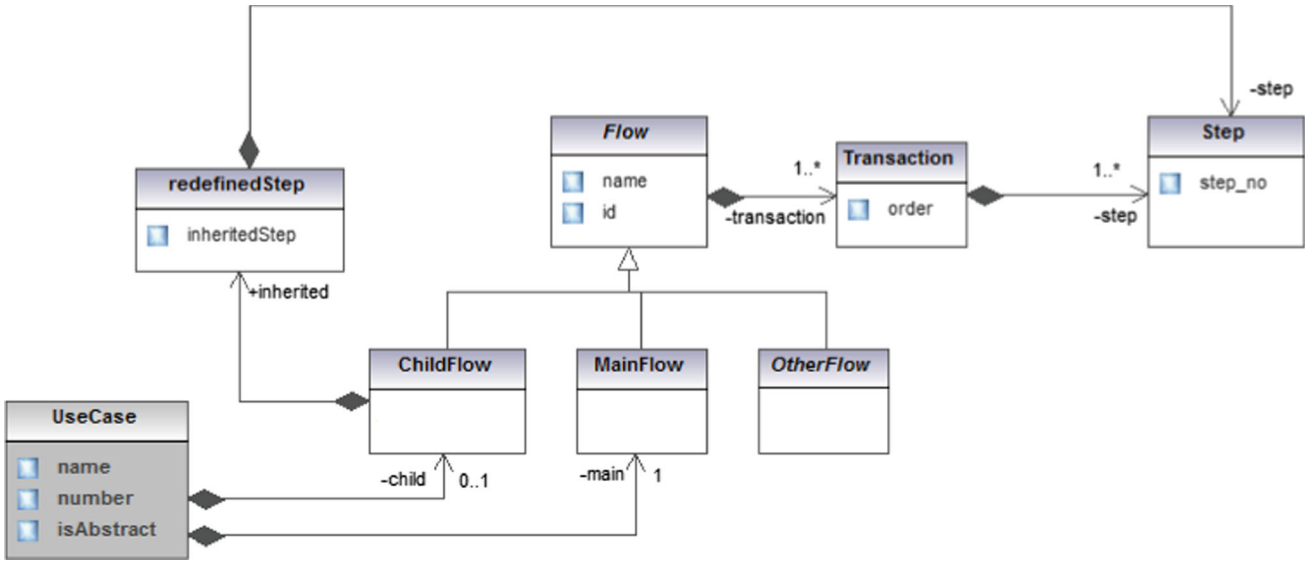


Fig. 12 Excerpt of the extended metamodel for the use case flow steps with generalization

Use Case: Reservation

Main Flow

1. The system displays a list of options available for reservation.
2. The customer selects an option.
3. The system displays the total cost.
4. The system displays the reservation confirmation number.
5. The use case ends

Use Case: Reserve Conference

Child Flow

1. super: 1
2. super: 2
   a. The customer selects to reserve a conference room.
3. The customer selects the room size, duration and additional equipment required
4. The system computes the cost.
5. super: 3
6. super: 4
7. super: 5

Fig. 13 A use case flow generalization example where the reservation use case generalizes the reserve conference use case

used as-is. Similar to the manner we handled alternative flow in Sect. 3.4.2, we define a new meta-class *redefinedStep*. This meta-class has a meta-attribute *inheritedStep* which references the step number inherited from the parent use case. Hence a child flow is composed of regular steps and redefined steps. A redefined step can be rewritten; hence, we add a relationship between the *inheritedStep* meta-class and the *Step* meta-class to facilitate this information. A modified version of the use case metamodel extension is depicted in Fig. 8 that handles use case generalization within its flow description is shown in Fig. 12.

Figure 13 shows exemplarily how the main flow of use case Reservation is redefined in the child use case Reserve Conference. We used the keyword "*super*" to differentiate between a regular step and inherited step within the child flow description. Hence, our proposed extension not only allows reusability of actions that do not require rewriting, but also allows child use case to modify actions inherited from parent use case flow.

3.5 Use case constraints

A use case model is composed of a number of constraints related to different model elements. We first briefly describe these constraints prior to defining the metamodel extension. Constraints within a use case model include

1. Precondition: preconditions indicate circumstances that must be true prior to the execution of the use case behavior. A precondition on a use case explains the state the system must be in for the use case to begin.
2. Post-condition: a post-condition indicates circumstances that must be true after execution of the use case behavior. A post-condition on a use case explains the state the system will be at the end of its execution. Based on the concept of alternate scenarios presented in Table 2, a use case can result in one of many states depending on the execution path (scenario) followed. Hence, a use case can have a single successful post-condition and multiple fail-

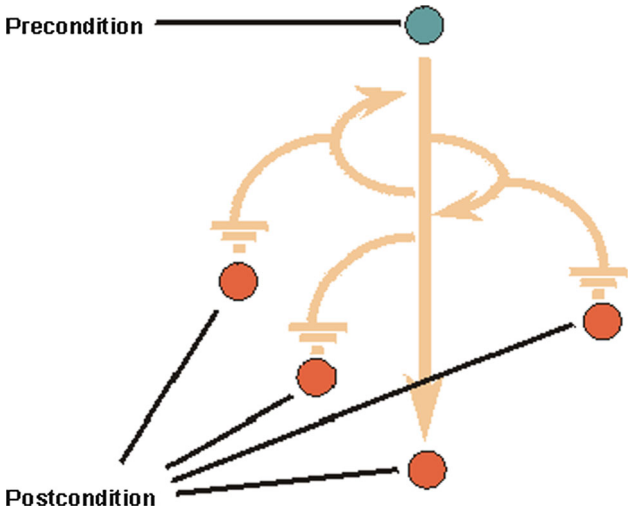


Fig. 14 Multiple use case scenarios adapted from [44]

ure or alternate post-conditions. This concept is explained appropriately by the illustration in Fig. 14 adopted from [44].

3. Extension/alternate flow constraint: execution of use case alternate flows or extension use cases require a condition to be satisfied prior to invoking the alternate flow or extension use case. This condition is referred to as a flow constraint.

All approaches that provide extensions to use case metamodel make use of a single meta-class called Constraint to handle use case constraints. Recent advancements in the field of use case modeling prompted the necessity of a structured storage and representation mechanism for constraints. Two main research proposals that make use of the use case constraint structure are (1) inferring use case sequencing relations from preconditions and post-conditions for requirements verification [57], use case synchronization [58], and test scenario generation [59], (2) enhancing software effort estimation process by assigning weights to preconditions, post-conditions, and constraints [60].

Prior to describing the use case metamodel extension with use case constraints, we included a meta-class in the metamodel called *Entity*. An entity, what most use case modeling tools refer to as Vocabulary or Glossary, refers to the systems under consideration, use cases, actors of the system, and their attributes. For instance, Customer and Transaction are entities of an ATM system use case model.

A use case constraint can either be atomic or compound. A compound constraint is composed of multiple atomic constraints constructed using Boolean operators (and, or and not). An atomic constraint is a 3-tuple $< E, R, V >$ where E is the entity, R is the relational operator, and V is the value. Values assigned to entities of the system can either be units such as *"logged in"* or numeric. For instance, a use case precondition "System is Active" can be written as $<$ System, $=$, Active $>$. In order to incorporate this structure in the use case metamodel, we add the following meta-classes: *Constraint, Atomic, Compound, Value, Relation, Numeric* and *Unit*. Figure 15 shows the excerpt of the use case constraint metamodel.

The complete extended use case metamodel is shown in Fig. 16.

4 Tool support

One of the vital functions of a metamodel description is to use it as a schema for semantic data that needs to be stored in a repository or model exchange. Hence, we have realized the metamodel described in this paper as part of a prototype tool called UCDesc (Use Case Description Tool) [61].

Experts agree that the most important aspect of use case analysis is the authoring of use case descriptions. Two foremost tools that are available for modeling the behavior aspect of use cases are CaseComplete [62] and Visual Use Case [63]. Both of these tools provide powerful features when it comes to composing use case flow of events. Although powerful use case editors, both the above-mentioned tools have some disadvantages. Visual Use Case lacks the functionality to export its models to XMI, a standard interchange format among UML CASE tools. Due to this limitation, output models cannot be reused for analysis in other tools. Although CaseComplete does provide an XMI export capability to the user, it lacks a fine-grained description of the flow of events and constraints in the resultant XMI file. In order to overcome these shortcomings, UCDesc provides the ability to export to XMI format so that it can be re-used by other UML modeling tools for analysis and integration. And since it is based on the metamodel proposed in this paper, it provides a fine-grained and effective manner of semantic data representation for storage and exchange. Apart from this, users can create a use case model, import use case structural model, add behavior to each use case, and evaluate the resultant model.

In order to make a use case description tool more amenable for the industrial practitioners, its logic engine should be concealed behind a graphical user interface. The initiative is that the user should be able to compose a use case description without constraints and the background engine handles the additional specifications (as defined by our metamodel) such as constraint analysis and model element identification. UCDesc is a simple use case description tool built in Java. The primary objective of UCDesc is to allow users to compose use case descriptions and provide the capability of exporting it to XMI. Figure 17 shows a view of UCDesc use case editing tool. The use case flows and constraints are then parsed using a Linguistic Parts-of-Speech Tagging Parser [64] and stored so that it conforms to the metamodel proposed in this

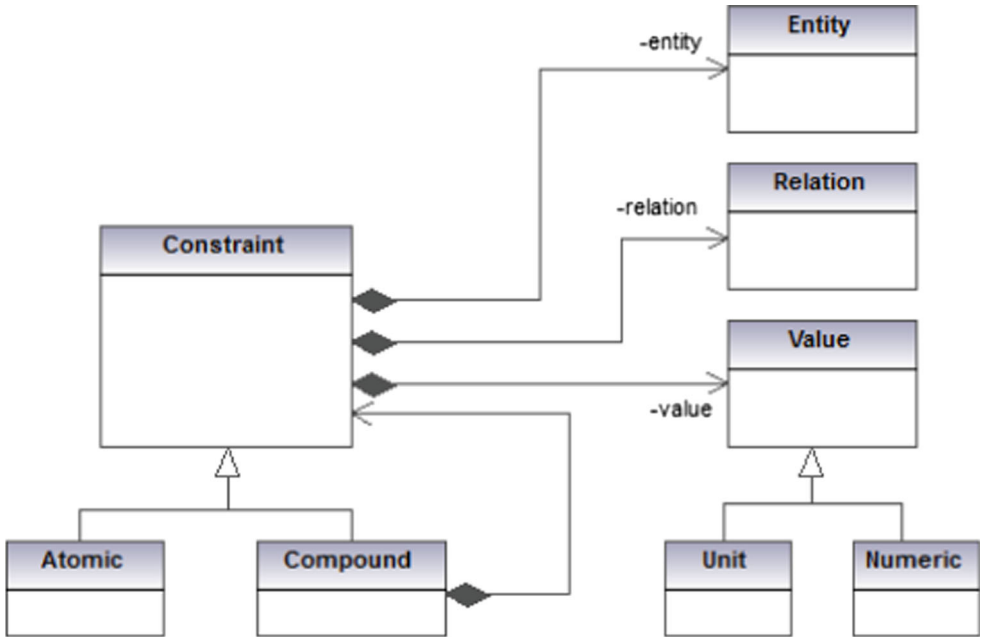


Fig. 15 Excerpt of the extended metamodel for constraint

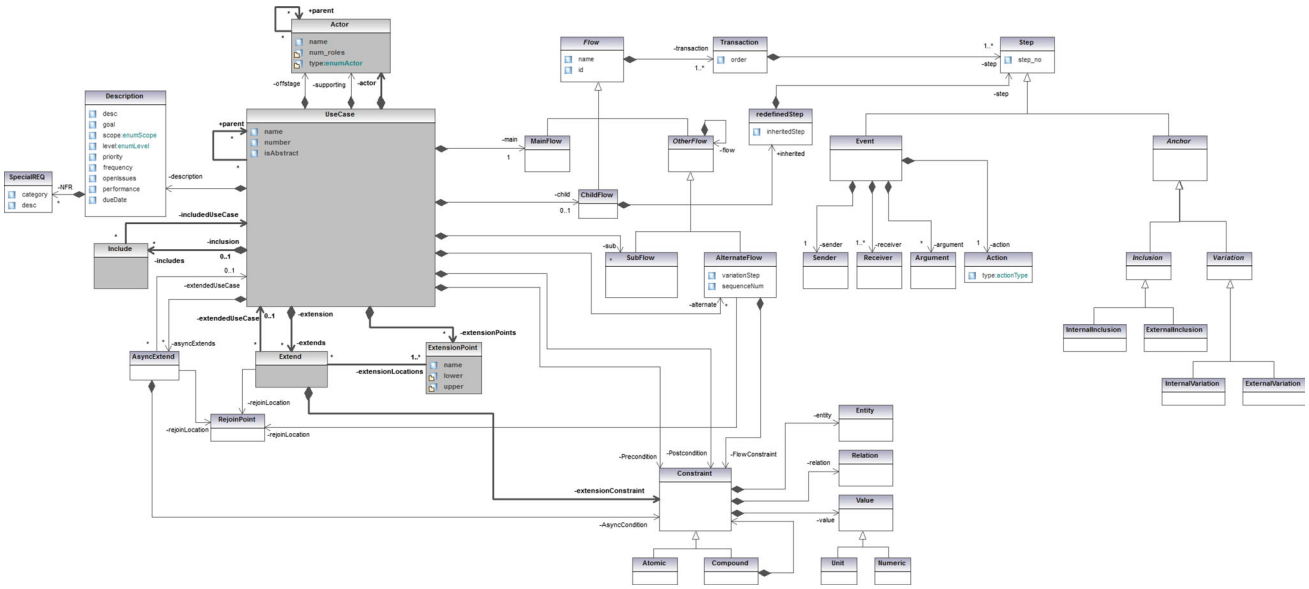


Fig. 16 The extended use case metamodel

a paper. The use case model then can be exported for further analysis and evaluation.

5 Applications of the proposed use case metamodel

This section illustrates how modeling is carried out in practice with the metamodel proposed in this paper. Two main applications of the proposed metamodel identified as a result of the extension are enabling use case-driven requirement analysis and model interchange. In the following subsections, we demonstrate how models based on the proposed metamodel can be used for the aforementioned applications.

5.1 Use case driven requirement analysis

Early detection of problems within requirements is an important issue as it impacts overall software quality and other project-related factors such as budget overruns and schedule slippage. Since use cases are primarily used to model the functional requirements of a software system, it is used as a

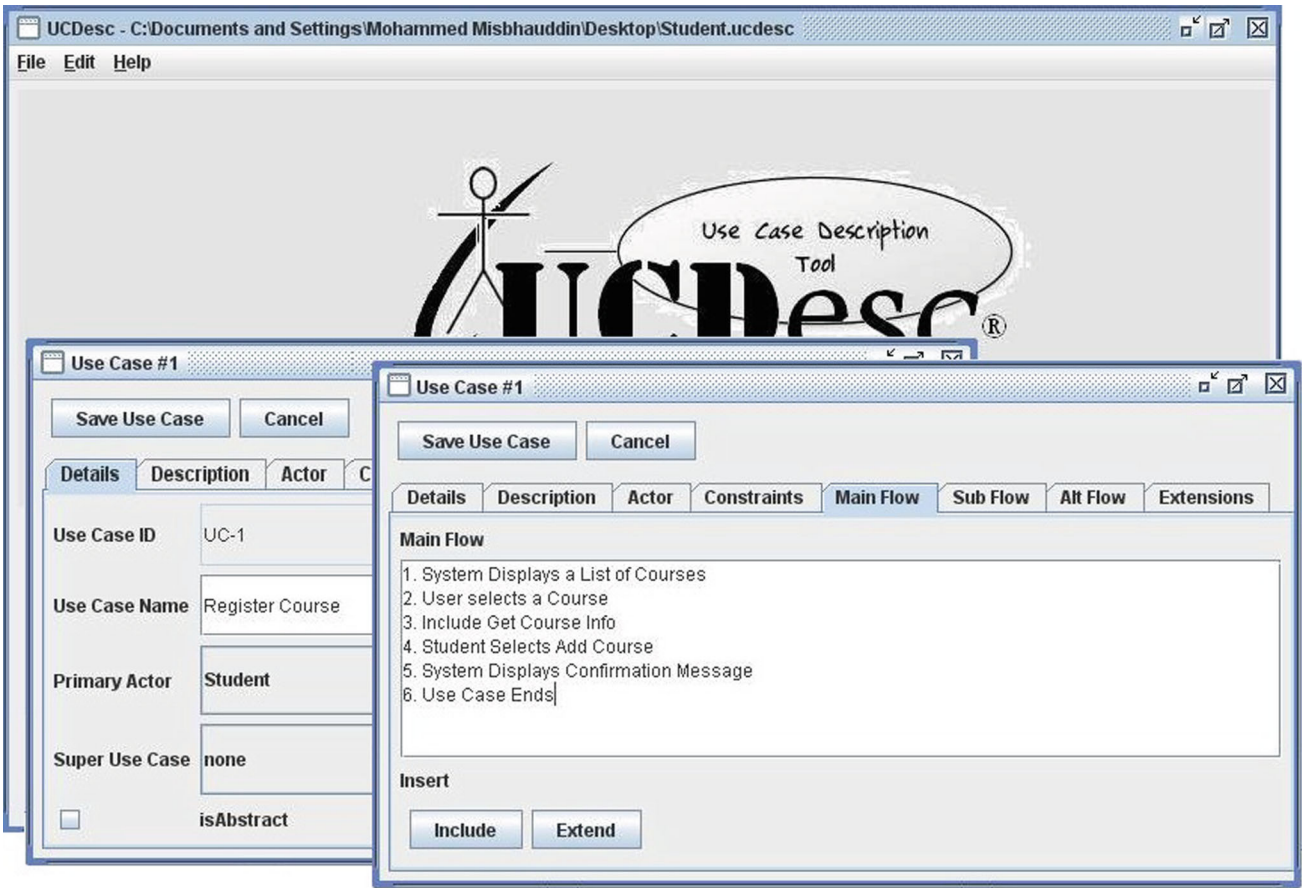


Fig. 17 UCDesc tool layout screenshot

key artifact for requirements quality assurance. An essential benefit of use case driven analysis is that it allows customers to actively participate in requirements analysis [65], identify and resolve conflicts in requirement drafts [66], and ensure consistency with customer and user intentions [67].

The extended use case metamodel captures functional and behavioral aspects of the system that are important for requirement analysis. In the following, we discuss the potential benefits of the extended metamodel in requirement analysis through verification and validation.

1. Test case generation: one of the ways in which the correctness of an implemented system is validated is through requirements traceability. Automatic generation of test cases is perhaps one of the most important advantages of the proposed extended metamodel. The use case constraint metamodel (part of the extended metamodel) provides a structured manner of representing the pre-conditions and post-conditions associated with a use case. A sequential relation can then be established between use cases by comparing pre-conditions and post-conditions. Finally, use case behavior is combined into a global control-flow based state model. Path sequences derived from this global model can then be used for system level testing [59].
2. Use case-driven requirements verification: use case metrics are most commonly used in literature to assess the quality of requirements. These measures are typically used to quantity the size and complexity of the requirements specification and use them as predictors of external quality attributes such as effort [68], defect-proneness [20], usability [69], and requirements importance [70] and so on. The extended use case metamodel not only provides a structure to define simple metrics such as number of actors and number of use cases (that can be also be done using the UML metamodel), but also complex metrics such as number of actor actions steps in a use case, number of system action steps in a use case, and so on. Definition of complex measures over the use case model is enabled through the low-level modeling of use case steps (into *Sender, Receiver, Action* and *Argument*) within the extended metamodel.

UML use case models do not enable effective requirements elicitation and analysis unless integrated with other text-based approaches [71] or formal approaches [72]. Exemplification of all the above-mentioned benefits of using the

Table 3 Factors for calculating the UUSP metric

Abbreviation	Name	Formula	Description
TPA	Total complexity of actor	ΣCA_i	CA = Data provided to & received from the use case
TPPrC	Total complexity of precondition	$\Sigma CPrC_i$	CPrC = # of logical expressions tested by a condition
PCP	Complexity of main scenario	PCA	PCA = # of entities + # of steps
TPCA	Total complexity of scenario	ΣPCA_i	
TPE	Total complexity of exception	ΣCE_i	CE = # of logical expressions tested to detect exception occurance
TPPoc	Total complexity of post-condition	$\Sigma CPoC_i$	CPoC = # of entities in the condition

extended use case model for requirement analysis is not feasible due to spatial limitation. Therefore, in this section, we demonstrate how a sample use case based on the proposed metamodel can be used for effort estimation.

The use case model is a relevant and valuable artifact for early project size measurement and estimating effort. An effort estimation technique proposed by Karner [68] is considered one of the basic estimating techniques for predicting effort based on use cases, known as UCP. Based on this technique, a number of modified techniques for effort estimation have been proposed. For the sake of illustration, we consider the use case size point (USP) metric proposed by Braz and Vergilio [60]. Also the illustration demonstrates calculation of the unadjusted use case size points (UUSP) only for simplicity.

The USP metric proposed in [60] overcomes the limitations of the original UCP metric by considering use cases written with an extended version [34] improving accuracy of the estimation. The metric is applied to each use case separately allowing estimation of time and cost of development of a particular use case and not that of the whole system. The formula used for obtaining the UUSP value is shown in (1) and its elements are explained in Table 3. Based on the values obtained for each factor, they are classified and values/weights assigned (based on complexity of the factor) that will be used in the formula for UUSP calculation in (1). The final value of USP for a use case is then obtained using the formula in (2). The term "scenario" used by the authors in [60] is synonymous to the concept of flows used in our proposed metamodel extension. For more information on weight assignment, technical factor (FTA), and environment factor (FAA), please refer to [60].

$$\begin{aligned} \text{UUSP} &= \text{TPA} + \text{TPPrC} + \text{PCP} + \text{TPCA} \\ &\quad + \text{TPE} + \text{TPPoc} \qquad (1) \\ \text{USP} &= \text{UUSP} * (\text{FTA} - \text{FAA}) \qquad (2) \end{aligned}$$

An example use case flow description and its corresponding XMI Specification conforming to the extended use case metamodel are shown in Fig. 18. We use this use case as an example to illustrate how it can be analyzed and be used for obtaining the UUSP metric.

Calculating the complexity of the precondition and post-condition involves checking the number of testable logical expressions within the precondition and post-condition. Since the proposed metamodel segregates each logical expression into the Constraint tag, simply counting the number of constraint elements within the preconditions and post-condition can help in evaluating its complexity value. The example use case has a precondition and post-condition with one logical expression; hence the complexity is "Simple". The complexity of preconditions and post-conditions cannot be obtained when using other metamodel proposed in literature as all expressions within these conditions (pre and post) are grouped under a single parent node.

Calculating the complexity of the main scenario involves summing the number of entities and steps within the main flow. The proposed metamodel records entities involved in a step within the Sender and Receiver tag. Simply counting the number of steps and the number of senders and receivers within each step provides the complexity value. The PCP value for the example use case is 13 and considered "Average" based on the weighing scale. Similarly, the complexity of the alternate flow is 3 and considered "Very Simple". These values cannot be obtained when using other metamodels as they consider *Step* as the lowest level of abstraction within a flow, thereby making it difficult to get the number of entities involved in each step.

Calculating the complexity of the actor requires the amount of data provided or received by the actor from the use case. In order to obtain this value, each *Step* within the use case flows needs to be examined. Whenever the actor appears as either the sender or the receiver, the number of arguments within that step are recorded and summed to get the value. The data value for the Customer actor for the example use case obtained from our proposed metamodel is 3 which makes it to be of "Average" Complexity. These values cannot be obtained from models created from other metamodels based on the same reason given above that the lowest level of abstraction is *Step*.

The algorithm used to calculate the UUSP metric value for a use case using the XMI schema provided by our proposed metamodel extension is shown in the following:

Algorithm UUSP Metric Calculation using Extended Use Case Metamodel

Input: UC: Use Case at UseCaseModel/UseCase in the XMI file

Output: UUSP: Unadjusted Use Case Point Metric for Use Case UC

Auxiliary Functions

name():- returns the name of the actor when passed its ID reference

classify-actor(), classify-pre(), classify-scenario() and classify-post():- matches the argument value with factor classification and returns the factor value. Argument/Value pairs can be found in [60]

```
1:  Initialize data=0; ArrayList Entity = Empty
2:  actor = name(UC@actor-ref)
3:  for main in UC/MainFlow/Transaction/Step
4:      if(main/Sender@name = actor OR main/Receiver@name = actor)
5:          data += 1
6:      if(main/Sender@name is not in Entity)
7:          Entity.add(main/Sender@name)
8:      if(main/Receiver@name is not in Entity)
9:          Entity.add(main/Receiver@name)
10:     main-step++
11: PCA-main = size(Entity) + main-step
12: PCP = classify-scenario(PCA-main)
13: Reset Entity ArrayList to Empty
14: for alt in UC/AlternativeFlow
15:     for step in alt/Transaction/Step
16:         if(main/Sender@name = actor OR main/Receiver@name = actor)
17:             data +=1
18:         if(main/Sender@name is not in Entity)
19:             Entity.add(main/Sender@name)
20:         if(main/Receiver@name is not in Entity)
21:             Entity.add(main/Receiver@name)
22:         alt-step++
23: PCA-alt = size(Entity) + alt-step
24: TPCA = classify-scenario(PCA-alt)
25: TPA = classify-actor(data)
26: for pre in UC/Precondition
27:     CPrC = count(pre/Constraint)
28: TPPrC = classify-pre(CPrC)
29: for exp in UC/AsyncExtend
30:     CE = count(exp/Constraint)
31: TPE = classify-pre(CE)
32: Reset Entity ArrayList to Empty
33: for posts in UC/Postcondition/Success/Constraint
34:     if (posts/Entity@name is not in Entity)
35:         Entity.add(posts/Entity@name)
36: for postf in UC/Postcondition/Failure/Constraint
37:     if (postf/Entity@name is not in Entity)
38:         Entity.add(postf/Entity@name)
39: CPoC = size(Entity)
40: TPPoC = classify-post(CPoC)
41: Set UUSP = sum(TPA, TPPrC, PCP, TPCA, TPE, TPPoC)
```

Data/Values are retrieved from the use case model using the XMI tree structure. The root node is "UseCaseModel" and all lower-level nodes are accessed using the "/" operator. Attributes of a particular node (mainly the name attribute) is accessed using the "@" operator. The main algorithm listed makes use of a few auxiliary functions not described in detail in this paper due to spatial limitations.

5.2 Use case model interchange

Data exchange possibilities enabled through the standardization of modeling technologies coupled with accelerated web-based developments have made shared software development a reality. In order to ensure effective sharing of modeling information between heterogeneous tools, use of standard industry-wide model interchange format is required. Since the interchange format supported by the extended metamodel is proprietary, it is necessary to discuss its usability with respect two types of tools: UML CASE tools and other prototype tools developed by researchers and practitioners.

The extended use case metamodel proposed in this paper does not conform to the UML meta-model. Hence, use case models conforming to the proposed extended metamodel

Use Case: Perform Transaction
UC-ID: 005

SCOPE: System
LEVEL: Primary Task
PRIORITY: High

ASYNCHRONOUS
Lost Connectivity
{System is not connected}
1. Display Error Message
2. Terminate User Session
3. Use Case Ends

PRECONDITIONS
System is Connected

ACTOR
PRIMARY
Customer
SUPPORTING
Bank System

MAIN FLOW
1. *Include* **Login**
2. System Displays a list of Transactions
3. Customer Selects Transaction
4. {Transfer}
5. {Pay Bill}
6. System displays Transaction Summary
7. Use Case Ends

ALTERNATIVE FLOW
6 (a) Customer Selects Print
1. The system sends the summary to the Printer
2. Return: 6

SUCCESS POST-CONDITION
Transaction is Successful

FAILURE POST-CONDITION
Transaction failed

```
<UseCaseModel>
  <Actor id="actor_0" name="Customer" type="Human" num_roles="1"/>
  <Actor id="actor_1" name="Bank System" type="NetworkSystem"
                                                num_roles="1"/>
  <UseCase actor-ref="actor_0" id="UC-005" name="Perform
                                        Transaction" isAbstract="false">
    <Supporting actor-ref="actor_1"/>
    <Description scope="System" level="PrimaryTask" Priority="high"/>
    <Precondition>
      <Constraint>
        <Entity name="System"/>

        <Relation name="equals"/>
        <Value name="Connected"/>
      </Constraint>
    </Precondition>
    <Postcondition>
      <Success>
        <Constraint>
          <Entity name="Transaction"/>
          <Relation name="equals"/>
          <Value name="Successful"/>
        </Constraint>
      </Success>
      <Failure>
        <Constraint>
          <Entity name="Transaction"/>
          <Relation name="equals"/>
          <Value name="Failed"/>
        </Constraint>
      </Failure>
    </Postcondition>
    <AsyncExtend name="Lost Connectivity" uc-ref="UC-013">
      <Constraint>
        <Entity name="System"/>
        <Relation name="not-equals"/>
        <Value name="connected"/>
      </Constraint>
    </AsyncExtend>
    <Include uc-ref="UC-001"/>
    <Extend uc-ref="UC-003" extPoint="Transfer"/>
    <Extend uc-ref="UC-004" extPoint="Pay Bill"/>
    <ExtensionPoint name="Transfer" lower="0" upper="1">
      <Constraint>
        <Entity name="transaction"/>
        <Relation name="equals"/>
        <Value name="transfer"/>
      </Constraint>
      <RejoinLocation step="6"/>
    </ExtensionPoint>
    <ExtensionPoint name="Pay Bill" lower="0" upper="1">
      <Constraint>
```

Fig. 18 An example use case flow description and its equivalent XMI

cannot be interpreted using UML CASE tools. In order to ensure compatibility between the proposed model interchange schema and import formats of UML CASE tools, we provide a model transformation framework (integrated and accessible from the UCDesc prototype tool). This framework allows the user to export the use case model conforming to the UML metamodel specification. The mapping framework is simplified based on our initial assumption (see Sect. 3) that the UML metamodel should remain as a subset after the proposed extension. The mapping framework transforms the extended model schema into the UML Use Case Schema based on the mapping rules presented in Table 4. The only purpose for using the exported model is to visualize the structural use case diagram within other UML CASE tools.

Use cases are considered a primary artifact for modeling functional requirements. Hence, it is used as an input for various applications such as requirement verification and validation [57,59], automated translation into analysis and design models [9–11,55], refactoring [73–75], and so on. Since there does not exist a standard interchange format for exchanging use case descriptions, quite a few metamodels and proprietary model interchange schemas have been proposed in literature [20,22,55]. One major application of our proposed interchange schema is that it models use case information down to the lowest level of abstraction. Hence, researchers and practitioners can utilize this format without the need to "re-invent the wheel" and concentrate on their application to their research domain. Hence, models created by our prototype tool can serve as the initial step for the other tool saving significant time and effort.

6 Related works

A number of criticisms and suggestions for the modifications of the UML use case metamodel have been proposed in literature. These modifications typically involve extension of the metamodel to incorporate its behavior.

EXTENSION POINTS
Transfer [transaction = transfer] : Transfer , return: 6
Pay Bill [transaction = pay] : PayBill , return: 6

```
                    <Entity name="transaction"/>
                    <Relation name="equals"/>
                    <Value name="pay"/>
                </Constraint>
                <RejoinLocation step="6"/>
            </ExtensionPoint>
    <MainFlow>
            <Transaction order="1">
                <Step step-no="1">
                    <ExternalInclusion  uc-ref="UC-001"/>
                </Step>
            </Transaction>
            <Transaction order="2">
                <Step step-no="2">
                    <Sender name="System"/>
                    <Receiver name="Customer"/>
                    <Action type="action" name="display"/>
                    <Argument name="transaction"/>
                </Step>
                <Step step-no="3">
                    <Sender name="Customer"/>
                    <Receiver name="System"/>
                    <Action type="action" name="select"/>
                    <Argument name="transaction"/>
                </Step>
                <Step step-no="4">
                    <ExternalVariation extPoint="Transfer"/>
                </Step>
                <Step step-no="5">
                    <ExternalVariation extPoint="Pay Bill"/>
                </Step>
                <Step step-no="6">
                    <Sender name="System"/>
                    <Receiver name="Customer"/>
                    <Action type="action" name="display"/>
                    <Argument name="tranbsaction summary"/>
                </Step>
            </Transaction>
    </MainFlow>
    <AlternativeFlow variationStep="6" sequence="a">
            <Constraint>
                <Entity name="selection"/>
                <Relation name="equals"/>
                <Value name="print"/>
            </Constraint>
            <Transaction order="1">
                <Step step-no="1">
                    <Sender name="System"/>
                    <Receiver name="Printer"/>
                    <Action type="action" name="send"/>
                    <Argument name="summary"/>
                </Step>
            </Transaction>
            <RejoinLocation step="6"/>
        </AlternativeFlow>
    </UseCase>
</UseCaseModel>
```

Fig. 18 continued

An extension to the UML metamodel for use cases was initially proposed by [20] for their XML-based requirements verification approach. They proposed a simple extension wherein a use case is composed of a sequence of steps. Each step refers to an optional condition, set of exceptions, or an action. The metamodel defined as part of their approach distinguished between different actions such as actor's actions (by the actor), system's actions (by the system), and use case actions (inclusions and extensions).

Rui and Butler [22] proposed a use case metamodel based on a single use case modeling notation. Elements in their metamodel are divided into three levels. Environmental level is similar to that of the UML use case metamodel which includes actors, use cases, and other feature-based information such as goals, services, and tasks. At the structural level, use case from the previous level is further decomposed into a series of episodes along with preconditions and post-conditions. In the event level, each episode from the structural level is further decomposed into events. An event is further classified as stimulus, response, or an action.

Diaz et al. [23] proposed a use case specification metamodel as an extension to the use case package of the UML metamodel. Each use case in their proposed extension includes a specification element which is composed of two different paths in a textual specification: basic and alternative. Each path is composed of a sentence which is classified as either a simple sentence or a special sentence («extends», «include» and control).

Metz et al. [21] did not propose an extension to the use case metamodel but provided an in-depth explanation of the different types of alternative flows in a use case description. They focused on unifying specific notational issues such as alterna-

Table 4 Mapping rules between extended UC Schema and UML UC Schema

Rule #	Extended Use Case Metamodel Schema	UML Use Case Metamodel Schema
1	<UseCaseModel id="a" name="b">	<uml:Model xmi:id="a " name="b">
2	<Actor id="a" name="b" type="c" num_roles="d" superActor="e"/>	<packagedElement xmi:type="uml:Actor" xmi:id="a" name="b"> <generalization xmi:type="uml:Generalization" xmi:id="f" general="e"/> </packagedElement>
3	<UseCase actor-ref="a" id="b" name="d" isAbstract="e" superUC="f" > </UseCase>	<packagedElement xmi:type="uml:UseCase" xmi:id="b" name="d" isAbstract="e"> <generalization xmi:type="uml:Generalization" xmi:id="g" general="f"/> </packagedElement> <packagedElement xmi:type="uml:Association" xmi:id="k"> <ownedEnd xmi:type="uml:Property" xmi:id="l" type="b"> <association xmi:idref="k"/> </ownedEnd>
		<ownedEnd xmi:type="uml:Property" xmi:id="m" type="a"> <association xmi:idref="k"/> </ownedEnd> <memberEnd xmi:idref="l"/> <memberEnd xmi:idref="m"/> </packagedElement>
4	<UseCase> <Include uc-ref="a"/> </UseCase>	<packagedElement xmi:type="uml:UseCase"> <include xmi:type="uml:Include" xmi:id="b" addition="a"/> </packagedElement>
5	<UseCase> <Extend uc-ref="a" extPoint="b"/> </UseCase>	<packagedElement xmi:type="uml:UseCase" > <extend xmi:type="uml:Extend" xmi:id="c" extendedCase="a"> <extension xmi:idref="b "/> </extend> </packagedElement>
6	<UseCase> <ExtensionPoint name="a" lower="b" upper="c"> <Constraint> <Entity name="d"/> <Relation name="e"/> <Value name="f"/> </Constraint> <RejoinLocation step="g"/> </ExtensionPoint> </UseCase>	<packagedElement xmi:type="uml:UseCase" xmi:id="k"> <extensionPoint xmi:type="uml:ExtensionPoint" xmi:id="h" name="a"> <ownedRule xmi:type="uml:Constraint" xmi:id="i" name="j"> <constrainedElement xmi:idref="k"/> <specification xmi:type="uml:LiteralString" xmi:id="l" value="d+c+f"/> </ownedRule> </extensionPoint> </packagedElement>

tive flow types in use case modeling. This concept of use case variability specification was later integrated into a use case metamodel extension proposed by Bragança and Machado [19]. They extended the use case metamodel with new model elements to clarify the use case relationships («extends», and «include»). The *Extend* meta-class from the UML metamodel was extended to include extension fragments. They associated a rejoin point (the return location within the base use case after execution of the extension fragment) with each extension fragment.

Hoffman et al. [24] recently proposed a narrative metamodel for textual use case descriptions specifying the behavior of use cases in a flow-oriented manner. The main motivation behind their approach was to ensure consistency between UML use case model and its descriptions. Each use case from the UML use case model is described as flow of events which is easily comprehendible by both technical and non-technical stakeholders.

Zelinka and Vranic [26] proposed a precise definition of different use case templates so as to allow a consistent application. Their goal was not to unify the UML use case model with its textual description, but to map the common and variable part among different template descriptions. This allowed flexibility of using a single notation or a combination of several use case description notations.

Somé [25] proposed a use case specification metamodel which is formally defined as an extension to the UML metamodel specification. He provided a set of constraints that ensure consistency between use case descriptions and use case models. He also enhanced the degree of expressiveness by introducing control flow structures for iteration and concurrency and definition of variable custom traits.

The most recent extension proposed for a use case metamodel was by Repond et al. [27]. Particularly they modeled the generalization relationship within a use case behavior which was not provided by earlier proposed extensions. They also defined the concept of use case scenarios that represents a specific path among all the possible flows of the use case. Hence each use case consists of multiple scenarios where each scenario has a sequence of steps that model a specific flow path. Although the concept of scenarios was put forward earlier by [22], it was not properly explained in their work.

It is also noteworthy to mention approaches that provide sentence-level analysis of use case descriptions. A number of approaches have been proposed for analyzing textual use case descriptions using natural language processing (NLP) techniques. These approaches can be broadly classified into two categories: lexical and semantic analysis. Lexical analysis approaches identify occurrences of certain patterns in use case descriptions [49,54]. Semantic analysis approaches identify aspects of a behavioral model. Some of the semantic analysis approaches make use of domain-specific verb categorization models [47,48,50–52] and others make use of configurable analysis engines that can be adapted to any domain and language [53,55].

Extensions proposed by [19–22] fail to include the concept of flows (or scenarios) within use case descriptions. Extensions proposed by [23–26] explicitly modeled flows as a set of steps within a use case description. Although these works modeled use case flow, the lowest level of abstraction in their work is a use case step of which a flow is composed of. In our proposed extension, we considered different form of steps within a use case flow and each action step further is modeled to the level of fine grain system-user interaction. Apart from this, we modeled the concept of use case transactions useful for effort estimation and use case analysis.

Almost all extensions proposed to the UML use case metamodel do not model the generalization relationship except for the metamodel proposed by Repond et al. [27]. Their work introduced a *GeneralizationPoint* where specialized use cases can add additional behavior. In our proposed extension, a specialized use case can not only add additional behavior, but can also modify or replace the steps of the generalized use case. Also the concept of *GeneralizationPoint* within the generalized use case defeats the purpose of generalization (*i.e.* allowing the generalized use case to have knowledge of what all use cases specializes it and where they add additional behavior).

Finally, all proposed extensions in literature cannot be used for use case analysis and evaluation due to lack of information modeled such as different actor types, use case transactions, and structure for use case constraints. In our proposed metamodel, we incorporated all required information for use case analysis and evaluation enhancing the usability of a use case model while maintaining the level of stakeholder comprehension.

7 Threats to validity

The main motivation of proposing an extension to the UML use case metamodel, apart from integrating its behavioral information, is to provide a complete metamodel. In pursuit of completeness, a number of use case templates, use case behavior composition methodologies, and their applications were considered. There are no standard use case templates available and new templates can be proposed in future to adapt to specific domain or application. Adapting the proposed metamodel to include novel elements might not be an easy task and may result in requiring complete metamodel restructuring. The main reason behind this threat is the manner in which this metamodel was constructed.

8 Conclusion and future work

In this paper we proposed an extension to the UML Use Case Metamodel. The proposed metamodel of use case diagram models both the structural view and behavioral view of the use cases, which can be used for model composition, model evaluation, and model interchange. The proposed metamodel was implemented as a prototype tool called UCDesc. Use case descriptions based on the proposed behavioral metamodel has a degree of completeness and accuracy needed to ensure consistency between the UML use case model and its textual descriptions, without constraining readability and understandability.

Our future works include adding formal description to the extended metamodel and define well-formedness rules based on this formal description. In order to measure and compare the usability and effectiveness of the proposed metamodel, we also plan to identify metrics to compare different metamodels and empirically study how the additional information added to the extended metamodel impacts its usage with respect to real-world applications. Since the proposed metamodel loses vital behavior information when imported into UML case tools, we plan to extend our work to propose a transformation framework to transform from the extended use case diagram to other UML diagrams such as activity, sequence, and state machine. We would also like to elaborate UCDesc by adding structural model composition using drawing tools. Furthermore, we would like UCDesc to evaluate the quality of the model by means of use case metrics and provide support for enhancing the quality by performing automated model smell detection and refactoring.

Acknowledgments This research work is partially supported by King Abdul Aziz City for Science & Technology (KACST) as well as the Deanship of Scientific Research of the King Fahd University of Petroleum and Minerals.

References

1. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1, 2003
2. Sawyer, P., Sommerville, I., Viller, S.: PREview: tackling the real concerns of requirements engineering. Technical report , Lancaster University, Lancaster (1996)
3. Chung, L., Leite, J.C.: On non-functional requirements in software engineering. In: Alexander, T.B., Vinay, K.C., Paolo, G., Eric, S.Y. (eds.) Conceptual Modeling: Foundations and Applications, pp. 363–379. Springer, Berlin (2009)
4. Yu, E.: Strategic Actor Modeling for Requirements Engineering: Modelling Your System Goals-The I* Approach. Technical Report, British Computer Society-Requirements Engineering Special Interest Group, London, UK (2005)
5. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Sci. Comput. Programm. **20**, 3–50 (1993)
6. Jacobson, I.: Object Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading (1992)
7. OMG: Unified Modeling Language: Superstructure. In: Version: 2.4.1 vol. formal/2011-08-06, Object Management Group (2011)
8. Phillips, C., Kemp, E., Kek, S.M.: Extending UML use case modelling to support graphical user interface design. In: Proceedings of the 13th Australian Conference on Software Engineering (2001)
9. Almendros-Jiménez, J.M., Iribarne, L.: Describing use-case relationships with sequence diagrams. Comput. J. **50**, 116–128 (2007)
10. Li, L.: Translating use cases to sequence diagrams. In: Proceedings of the 15th IEEE International conference on Automated Software Engineering, pp. 293–296. Grenoble, France (2000)
11. Yue, T., Briand, L. C., Labiche, Y.: Automatically deriving UML sequence diagrams from use cases. Technical Report, Carleton University, Canada, TR-SCE-10-03 (2010)
12. Gutiérrez, J., Nebut, C., Escalona, M., Mejías, M., Ramos, I.: Visualization of use cases through automatically generated activity diagrams. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) Model Driven Engineering Languages and Systems, vol. 5301, pp. 83–96. Springer, Berlin/Heidelberg (2008)
13. Lei, M., Jiang, W.C.: Research on activity based use case metamodel. In: International Conference on Advanced Computer Theory and Engineering, pp. 843–846 (2008)
14. Nakatani, T., Urai, T., Ohmura, S., Tamai, T.: A requirements description metamodel for use cases. In: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference, p. 251 (2001)
15. Back, R.-J., Petre, L., Porres, I.: Formalising UML use cases in the refinement calculus. Technical Report, Turku Centre for Computer Science, Turku, Finland, TUCS-TR-279 (1999)
16. Dranidis, D., Tigka, K., Kefalas, P.: Formal modelling of use cases with X-machines. In: Proceedings of the 1st South-East European Workshop on Formal Methods (2003)
17. Grieskamp, W., Lepper, M., Schulte, W., Tillmann, N.: Testable use cases in the abstract state machine language. In: Proceedings of the Second Asia-Pacific Conference on Quality Software, Washington, DC, p. 167 (2001)
18. Grieskamp, W., Lepper, M.: Using use cases in executable Z. In: Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods, Washington, DC, USA, p. 111 (2000)
19. Bragança, A., Machado, R.J.: Extending UML 2.0 Metamodel for complementary usages of the << extend >> relationship within use case variability specification. in 10th International Software Product Line Conference, Baltimore, USA, 2006, pp. 123–130
20. Durán, A., Ruiz-Cortés, A., Corchuelo, R., Toro, M.: Supporting requirements verification using XSLT. In: IEEE International Requirements Engineering Conference, Germany, Essen, pp. 165–172 (2002)
21. Metz, P., O'Brien, J., Weber, W.: Specifying use case interaction: types of alternative courses. J. Object Technol. **2**, 111–131 (2003)
22. Rui, K., Butler, G.: Refactoring use case models: the metamodel. In: Proceedings of the 25th Australasian Computer Society Conference, pp. 301–308 (2003)
23. Díaz, I., Losavio, F., Matteo, A., Pastor, O.: A specification pattern for use cases. J. Inform. Management **41**, 961–975 (2004)
24. Hoffmann, V., Lichter, H., Nyáen, A., Walter, A.: Towards the integration of UML and textual use case modeling. J. Object Technol. **8**, 85–100 (2009)
25. Somé, S.S.: A meta-model for textual use case description. J. Object Technol. **8**, 87–106 (2009)
26. Zelinka, L.u., Vrani'c, V.: A configurable UML based use case modeling metamodel. In: First IEEE Eastern European Conference on the Engineering of Computer Based Systems, Washington, DC, pp. 1–8 (2009)
27. Repond, J., Dugerdil, P., Descombes, P.: Use-case and scenario metamodeling for automated processing in a reverse engineering tool. In: 4th India software engineering conference, New York, pp. 135–144 (2011)
28. OMG: XML Metadata Interchange Specification 2.1.1, vol. formal/2007-12-01, ed: Object Management Group, 2007
29. OMG: Meta Object Facility (MOF), vol. formal/2006-01-01, ed: Object Management Group (2006)
30. Cockburn, A.: Writing effective use cases. Addison-Wesley, Reading (2000)
31. Kruchten, P.: The Rational Unified Process: An Introduction, vol. 2. Addison-Wesley, Reading (2000)
32. Leite, J.C., Doorn, J., Hadad, G., Doorn, J.H., Kaplan, G.: A scenario construction process. Requir. Eng. J. **5**, 38–61 (2000)
33. Toro, A.D., Jiménez, B.B., Cortés, A.R., Bonilla, M.T.: A requirements elicitation approach based in templates and patterns. In: Workshop em Engenharia de Requisitos, pp. 17–29 (1999)
34. Larman, C.: Applying UML and patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, vol. 2. Prentice Hall PTR, Upper Saddle River (2001)
35. Schneider, G., Winters, J.P.: Applying use cases: a practical guide, vol. 2. Addison-Wesley, Reading (2001)
36. Nunes, N.J.: iUCP-estimating interaction design projects with enhanced use case points. In: England, D., Palanque, P., Vanderdonckt, J., Wild, P. (eds.) Task Models and Diagrams for User Interface Design, vol. 5963, pp. 131–145. Springer, Berlin/Heidelberg (2010)
37. Constantine, L.L., Lockwood Lucy, A.D.: Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design. Addison Wesley, Longman (1999)
38. Adolph, S., Bramble, P., Cockburn, A., Pols, A.: Patterns for Effective Use Cases. Addison-Wesley, Reading (2003)
39. Armour, F., Miller, G.: Advanced Use Case Modeling: Software Systems. Addison-Wesley, Reading (2001)
40. Arlow, J., Neustadt, I.: UML and the Unified Process: Practical Object-Oriented Analysis and Design. Addison-Wesley, Reading (2002)
41. Bittner, K., Spence, I.: Use Case Modeling. Addison-Wesley, Reading (2003)
42. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, vol. 2. Addison-Wesley, Reading (2005)
43. Hilsbos, M., Song, I.-Y., Choi, Y.: A comparative analysis of use case relationships. In: Akoka, J., Liddle, S., Song, I.-Y., Bertolotto, M., Comyn-Wattiau, I., van den Heuvel, W.-J., et al. (eds.) Perspectives in Conceptual Modeling, vol. 3770, pp. 53–62. Springer, Berlin/Heidelberg (2005)

44. Constantine L.L., Lockwood Lucy, A.D.: Structure and style in use cases for user interface design. In: van Harmelen, M. (ed.) Object Modeling and User Interface Design. Addison-Wesley: Reading (2001)
45. Laguna, M.A., Marqués, J.M.: On the Multiplicity Semantics of the Extend Relationship in Use Case Models. In: Cordeiro, J., Shishkov, B., Ranchordas, A., Helfert, M. (eds.) Software and Data Technologies, vol. 47, pp. 62–75. Springer, Berlin/Heidelberg (2009)
46. Diev, S.: Software estimation in the maintenance context. SIGSOFT Softw. Eng. Notes **31**, 1–8 (2006)
47. Ambriola, V., Gervasi, V.: On the systematic analysis of natural language requirements with CIRCE. Automated Softw. Eng. **13**, 107–167 (2006)
48. Fliedl, G., Kop, C., Mayr, H.C., Salbrechter, A., Vöhringer, J., Weber, G., et al.: Deriving static and dynamic concepts from software requirements using sophisticated tagging. J. Data Knowl. Eng. **61**, 433–448 (2007)
49. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Application of linguistic techniques for use case analysis. In: International Requirements Engineering Conference, London Limited, pp. 161–170 (2003)
50. Overmyer, S., Lavoie, B., Rambow, O.: Conceptual modeling through linguistic analysis using LIDA. In: 23rd International Conference on Software Engineering, Washington, pp. 401–410 (2001)
51. Rolland, C., Ben Achour, C.: Guiding the construction of textual use case specifications. J. Data Knowl. Eng. 25, 125–160 (1998)
52. Sampaio, A., Chitchyan, R., Rashid, A., Rayson, P.: EA-Miner: a tool for automating aspect-oriented requirements identification. In: International Conference on Automated Software Engineering, New York, 2005, pp. 352–355
53. Sinha, A., Paradkar, A. M., Kumanan, P., Boguraev, B.: A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In: International Conference on Dependable Systems and, Networks, pp. 327–336 (2009)
54. Wilson, W.M., Rosenberg, L.H., Hyatt, L.E.: Automated analysis of requirement specifications. In: 19th International Conference on Software Engineering, New York, pp. 161–171 (1997)
55. Yue, T., Briand, L.C., Labiche, Y.: Automatically Deriving a UML Analysis Model from a Use Case Model. Technical Report, Carleton University, Canada, TR SCE-09-09 (2010)
56. Ochodek, M., Nawrocki, J. R.: Automatic transactions identification in use cases. In: Second Central and East European Conference on Software Engineering Techniques, pp. 55–68 (2011)
57. Somé, S.S., Nair, D.K. : Use case based requirements verification–verifying the consistency between use cases and assertions. In: 9th International Conference on, Enterprise Information Systems, pp. 190–195 (2007)
58. Somé, S.S.: Specifying use case sequencing constraints using description elements. In: Sixth international workshop on scenarios and state machines, Washington, DC, pp. 4–10 (2007)
59. Somé, S.S., Cheng, X.: An approach for supporting system-level test scenarios generation from textual use cases. In: ACM Symposium on Applied computing, New York, pp. 724–729 (2008)
60. Braz, M.R., Vergilio, S.R.: Software effort estimation based on use cases. In: 30th Annual International Computer Software and Applications Conference, pp. 221–228. Washington, DC (2006)
61. Misbhauddin, M., Alshayeb, M.: UCDesc, vol. 2012. King Fahd University, Dhahran (2011)
62. Serlio: "CaseComplete". Software Development Corporation (2011)
63. TechnoSolutions: Visual Use Case (2009)
64. Toutanova, K., Klein, D., Manning, C., Singer, Y.: Stanford POS Tagger, 3.0.4 ed. Stanford University, NLP Group, Stanford (2003)
65. Regnell, B., Andersson, M., Bergstrand, J.: A Hierarchical Use Case Model with Graphical Representation. In: Proceedings of the IEEE Symposium and Workshop on Engineering of Computer Based Systems (1996)
66. Kotonya, G., Sommerville, I.: Requirements Engineering: Processes and Techniques. Wiley, London (1998)
67. Bernárdez, B., Durán, A., Genero, M.: Empirical evaluation and review of a metrics-based approach for use case verification. J. Res. Pract. Inform. Technol. **36**, 247–258 (2004)
68. Karner, G.: Metrics for Objectory. University of Linkoping, Sweden, Diploma Thesis (1993)
69. Hornbæk, K., Høegh, R.T., Pedersen, M.B., Stage, J.: Use case evaluation (UCE): a method for early usability evaluation in software development. In: Proceedings of the 11th IFIP TC 13 international conference on Human-computer interaction, Rio de Janeiro, Brazil (2007)
70. Kim, H., Boldyreff, C.: Developing software metrics applicable to UML models. In: 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented, Software Engineering (2002)
71. Kim, J., Park, S., Sugumaran, V.: Improving use case driven analysis using goal and scenario authoring: a linguistics-based approach. Data. Knowl. Eng. **58**, 21–46 (2006)
72. Sinha, A., Kaplan, M., Paradkar, A., Williams, C.: Requirements modeling and validation using bi-layer use case descriptions. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) Model Driven Engineering Languages and Systems, vol. 5301, pp. 97–112. Springer, Berlin/Heidelberg (2008)
73. Rui, K.: Refactoring use case models. PhD Thesis, Concordia University (2007)
74. Kim, Y., Doh, K.-G.: The service modeling process based on use case refactoring. In: Abramowicz, W. (ed.) Business Information Systems, vol. 4439, pp. 108–120. Springer, Berlin/Heidelberg (2007)
75. Issa, A.: Utilising refactoring to restructure use-case models. In: Proceedings of International Conference of Information Engineering, London, pp. 523–527 (2007)

Author Biographies

Mohammed Misbhauddin is an assistant professor at King Faisal University, Saudi Arabia. He received his MS in Computer Science from Illinois Institute of Technology, Chicago and his PhD in Computer Science and Engineering from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia. His research interests include model-driven software development, software refactoring, meta-modeling, software metrics and quality, artificial intelligence, and data mining.

Mohammad Alshayeb is an associate professor in Software Engineering at the Information and Computer Science Department, King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia. He received his MS and PhD in Computer Science and certificate of Software Engineering from the University of Alabama in Huntsville. He worked as a senior researcher and Software Engineer and managed software projects in the United States and Middle East. He is a certified project manager (PMP). His research interests include Software quality, software measurement, and metrics, and empirical studies in software engineering.