Hindawi
Journal of Engineering
Volume 2018, Article ID 6854920, 9 pages
https://doi.org/10.1155/2018/6854920

*Research Article*

# A Multiview Formal Model of Use Case Diagrams Using Z Notation: Towards Improving Functional Requirements Quality

**Khadija El Miloudi and Aziz Ettouhami**

*LCS Laboratory, Faculty of Sciences of Rabat, Mohamed V University in Rabat, Morocco*

Correspondence should be addressed to Khadija El Miloudi; elmiloudi.khadija@gmail.com

Received 5 June 2018; Revised 7 November 2018; Accepted 12 November 2018; Published 2 December 2018

Academic Editor: Lucian Dascalescu



We propose a new formal model of UML use case diagram using Z notation to address some of its shortcomings. UML use case diagram has therefore become commonly used to structure functional requirements and the greatest challenge facing the software developer nowadays is to deliver a high quality product meeting customers' requirements. However, the major disadvantage of UML models is their imprecision. In addition, they are basically in a form of semiformal modelling representations and associated natural language requirements and lack any mechanism to rigorously check consistency which results in its models being subject to ambiguity. This paper reports on the first formal modelling approach of use case diagrams in a multiview context. The approach is divided into two steps. In the first step, a formal model of UML use case diagram is proposed using Z notation. Then, a multiview consistency checking is presented. This approach guarantees software consistency, improving requirements quality.

## 1. Introduction

Today, people's daily life is strongly dependent on software systems which become increasingly large and complex. "In software, requirements are the foundation for everything we do, so the quality of requirements is paramount" [1]. As the software in today's systems grows larger, requirements documents become enormous and ambiguity is inevitably introduced. Many software development projects fail due to problems in requirements. Accordingly, it has become very important to be able to improve software quality by paying more attention to requirements activities. UML use case diagrams [2, 3] have therefore become commonly used during early stages of deriving software requirements and have been incorporated into most major object-oriented development methods thanks to its simplistic designs promoting good software engineering practices. However, on the one hand, the lack of formality in UML use case diagrams definition provides ample room for misunderstandings. On the other hand, UML models are not directly analysable and lack the ability to be type checked mechanically.

In this article, we propose a formal model of use case diagrams using Z notation [4, 5] to benefit from the rigour that comes from formal methods. By this formalism concept, we overcome UML use case diagram limitations allowing the ability to use type checkers and theorem provers on Z specification such as Z/EVES system. It is noteworthy that this paper reports on the first formalization of use case diagrams in Z notation that considers the consistency in multiview modelling.

This article is a continuation of [6–8], where they first described a formal model of class, sequence and state machine diagrams and checked the consistency between the different views. The extensions in the current paper are as follows: (1) a formal model of use case diagrams using Z notation, (2) a multiview consistency checking by means of Z theorems. As was pointed out above, "Formal specifications can be very useful artifacts at various stages of software development" [9]. The purpose of this article is to profit from the expressiveness and the mathematical basis of formal methods to improve requirements quality. More than this, "formal methods contribute to demonstrably cost-effective development of software with very low defect rate" [10].

The rest of this paper is organized as follows. Section 2 reviews relevant existing work that has been done in use case diagrams formalization. Section 3 presents a formal model

of use case diagrams using Z notation. Section 4 discusses multiview consistency between class, sequence and use case diagrams using Z theorems. Finally, Section 5 contains some concluding remarks and outlines some future directions of our work.

## 2. Related Work

Formalizing UML use case diagrams has drawn great attention from research community because it provides a considerable potential to help developers analyze whether the specification capture the intended requirements. In this section, we present some of the research literature showing approaches that define UML use case diagram semantics and discuss its similarity to and difference from our approach. Two main research schools have been working on this problem. The first is to propose techniques and tools to transform use case diagrams into different formalisms. For example, Junior et al. [11] propose a formalization of use cases using graph transformation models with the goal of running tool supported analyses on them and revealing possible errors. Sinnig et al. [12] propose an integrated development methodology based on non deterministic finite state machine formalism which defines a common semantics for use cases and task models. Butler et al. [13] define the notion of use case and the related terminology by means of a specification written in Z. The focus is on formalizing and clarifying the basic notions of use case and its related concepts. In our work, the focus is on specifying the different components and constraints imposed on/by use cases. Also formalizations of actors and inheritance relation between them and extension points are added. Barajas [14] presents the formal specification for a tool that models the functional requirements of a system based on use case models using Alloy. Our model is similar to that work but using Z language and with slight differences in addition to our multiview consistency checking. Scandurra et al. [15] propose the framework asmetaRE to automatically transform use cases models into ASM executable specifications and then validate systems requirements through simulation and scenario based simulation of the generated ASM. Shen and Liu [16] propose a new rigorous review technique including execution and testing which can be applied to software requirements models using a new language HCL (high level constraint language). Murali et al. [17] describe an approach for embedding a formal method within UML use case modeling using event-B. They extend use case modeling to allow for the explicit representation of safety concerns. Mostafa et al. [18] propose a formalization of the syntax of the popular UML diagrams (use case diagram, class diagram and state machine diagram) using Z specification. They formalize each basic notion of use case diagram without specifying it as a whole. In our model, the use case diagram is well explained and formalized. The model was improved by including the notion of scenario in case of extension.

While the first school has only targeted formalization issues, the second school focuses on multiview consistency between use case diagram and other UML diagrams. For example, Yue et al. [19, 20] propose a method and a tool called aToucan to automatically generate a UML analysis model comprising class, sequence and activity diagram from use case model. However, a use case diagram is not sufficient in its information to generate complete UML analysis model. While they chose to transform use case model into a UML analysis model, we propose a formal specification of use case diagram using Z notation providing a basis for checking consistency between use case diagram and a subset of formalized UML diagrams. Giese and Heldal [21] present a way to relate informal requirements in form of use cases to more formal specifications written in OCL. Ibrahim et al. [22] propose a set of consistency rules between use case diagram, sequence diagram and class diagram. The abstract syntax of the proposed consistency rules are formally defined using logical approach. Most of the consistency rules cited in this work are treated by our proposed model using Z notation.

Overall, these studies highlight the need for formalizing use case diagrams. However such studies remain narrow in their focus. Our approach includes transforming use case diagram into formal model using Z notation and multiview consistency checking between different UML diagrams. Furthermore, our work is one of very few existing works on multiview consistency checking and the first one using Z notation. Needless to say, through our approach both customers and developers can enjoy the benefits of formal languages to improve the quality of requirements and subsequently the quality of the obtained software.

## 3. Formal Model of Use Case Diagrams

Use case diagrams are considered for high level requirement analysis of a system for the purpose of capturing its dynamic view. They are essentially used to gather requirements and functionalities of a system captured in use cases and also to identify internal and external agents interacting with the system. These agents are known as actors. Figure 1 shows an example of use case diagram of ATM System [3].

So in brief, use case diagrams consist of actors, use cases and their relationships. Figure 2 illustrates the abstract syntax of use case diagrams [3].

The first step in the approach is to formalize the notion of use cases and relationships among them. To get a better understanding of the proposed formal model, we first define the different terms involved in the specification such as included use cases, extending use cases and extension point.

We will use Z notation to specify and describe the different notions of use case diagram. The basic concepts of Z notation [4, 5] are summarized below.

*3.1. Summary of Z Notation.* Z [4] is a formal specification language created by J.R. Abrial based upon set theory and mathematical logic. In Z notation, a specification uses the notion of schema to structure the underlying mathematics and allow an easy reuse of its subparts. Its graphical notation, called schema, is represented as below:

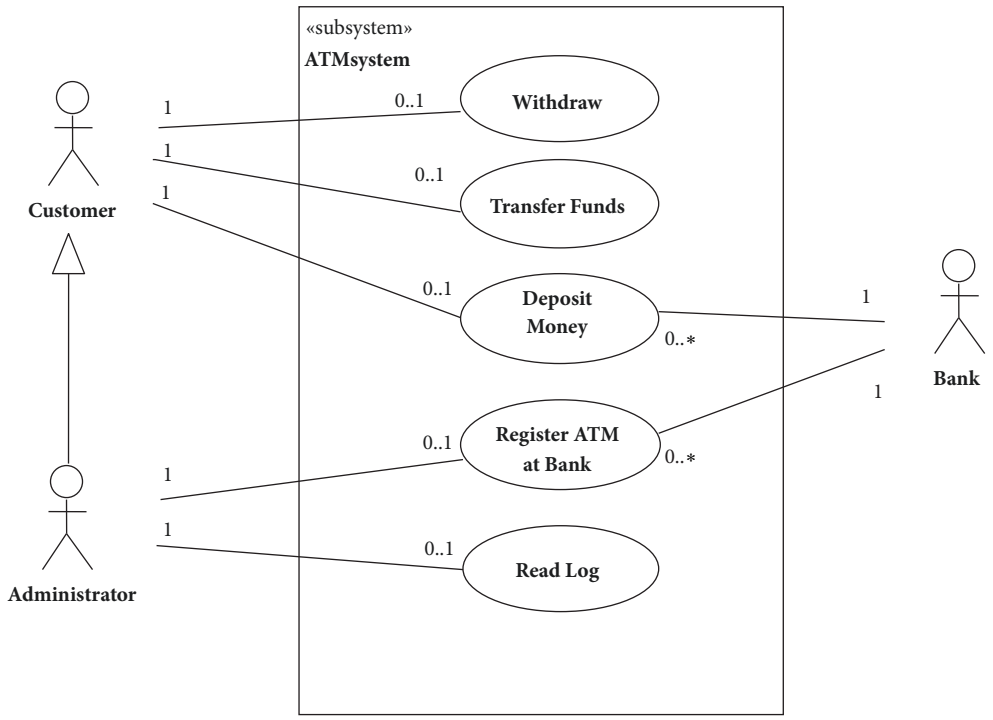


Figure 1: Example of the use cases and actors for an ATM system.

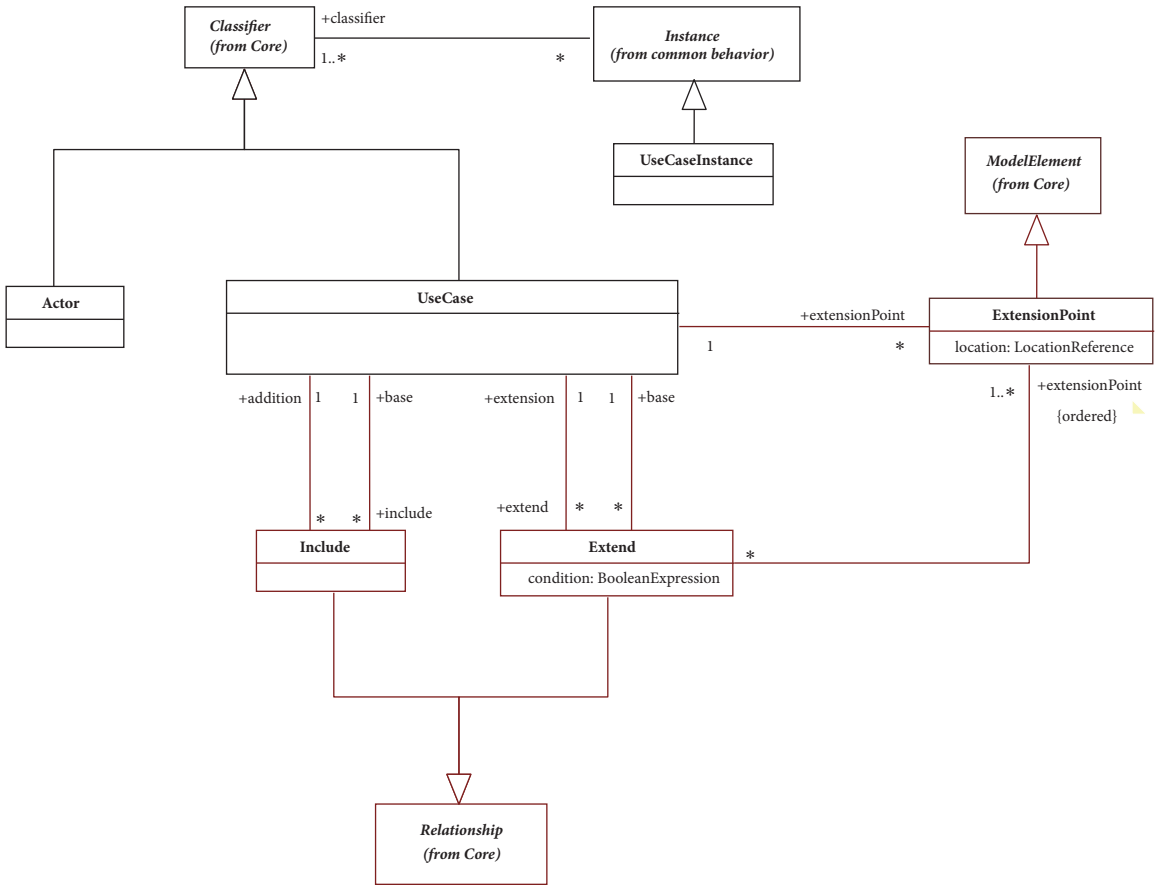


Figure 2: The abstract syntax of use cases package.

*Schema_Name*

*Declarations*

*Predicates*

A schema contains a declaration part that may contain the declaration of state variables beside references to other schemas and a predicate part which consists in a set of predicates constraining the variable state values. These predicates express properties on the state variables and introduce relationships between them. The name of the Z schema enables its re-use. A Z schema may be used or re-used as a declaration, a type or a predicate. When the specification requires a composite type, a schema is used to denote it.

*3.2. Use Cases Formalization.* According to the UML standard, a use case is defined as a classifier. Actually the set of all unique identifiers throughout the specification is denoted by the given set *CLASSIFIER*.

Therefore, the variable *UCASE* is introduced as a classifier to specify the set of use cases' identifiers.

$[CLASSIFIER]$

$UCASE: \mathbb{P}\ CLASSIFIER$

There are several different kinds of relationships between use cases. These are the include relationship, the extend relationship and generalization, all of which are described in this section.

Include is a relationship between two use cases which is used to show "that behavior of the included use case is inserted into the behavior of the including use case" [3]. Extend is a relationship "that specifies how and when the behavior defined in usually optional extending use case can be inserted into the behavior defined in the extended use case" [3].

The relationships include and extend are basically represented as a Z relation that relates two use cases. The relation *include* is used to record which use case is directly included by the original use case, whereas the relation *extend* is used to record which use case is directly extending the original use case.

The functions *includedUsecases* and *extendingUsecases* are respectively obtained with the transitive closure of *include* and *extend*. They are used to formally specify the function that returns the set of use cases that are directly or indirectly related with the relations include or extend.

$include: UCASE \leftrightarrow UCASE$
$extend: UCASE \leftrightarrow UCASE$
$includedUsecases: UCASE \rightarrow \mathbb{P}\ UCASE$
$extendingUsecases: UCASE \rightarrow \mathbb{P}\ UCASE$

---

$\forall u1: UCASE \bullet includedUsecases\ u1 = \{\ u2: UCASE \mid (u1, u2) \in include^{+}\ \}$
$\forall u1: UCASE \bullet extendingUsecases\ u1 = \{\ u2: UCASE \mid (u2, u1) \in extend^{+}\ \}$

The extension may occur at a specific point of the extended use case called extension point. An extension point is associated with a constraint indicating when the extension occurs. According to the UML standard, an extension point must have a name. Formally, an extension point is specified as follows:

*ExtensionPoint*

$name: NAME$
$constraint: \mathbb{P}\ BOOL$

---

$\#\ constraint \leqslant 1$

If no constraint is associated, the extension is unconditional. In this case, the state variable constraint is an empty set and therefore its cardinality is equal to 0.

A use case generalization is a relationship from a child use case to a parent use case, specifying how a child can specialize all behavior and characteristics described for the parent. A use case generalization is translated into Z by adding, in the schema *Usecase*, the state variable *parents* defined as a set of use cases.

Once the related terminology of use cases has been clarified by means of specification written in Z, the notion of use case may thereafter be fully formalized.

Therefore, the notion of use case is clearly represented in the following schema:

*Usecase*

*self: UCASE*
*name: NAME*
*parents:* $\mathbb{P}$ *UCASE*
*Fragments:* seq *FRAGMENT*
*includedUC:* $\mathbb{P}$ *UCASE*
*extendingUC:* $\mathbb{P}$ *UCASE*
*extendedUC:* $\mathbb{P}$ *UCASE*
*isExtended: BOOL*
*isExtending: BOOL*
*extensionPoints:* seq *ExtensionPoint*

---

$includedUC = includedUsecases\ self$
$self \notin includedUC \cup extendingUC$
**if** $isExtended = True$
**then** $extendingUC = extendingUsecases\ self$
**else** $extendingUC = \varnothing \wedge extensionPoints = \langle\rangle$

The top half of the *Usecase* schema defines the different variables of a use case with their respective types. Each use case is identified by its identifier called *self* and its name. As explained before, the variable *parents* is used to define the set of use cases specialized by the current use case.

If a use case is extended, it is then composed of a sequence of fragments and an ordered list of extension points. An extending use case consists of one or more behavior fragments descriptions that are to be inserted into the appropriate spots of the extended use case. The extension points specify where the respective behavioral fragments of the extending use case are to be inserted [3]. If the condition is true when the first extension point is reached, then the extending use case will be executed. Otherwise the extension does not occur. If no constraint is associated with the extend relationship, the extension is unconditional.

As stated in the first predicate in the second half of the schema *Usecase*, a use case cannot include use cases that directly or indirectly include it.

After having defined the notion of use case and its related terminology, we are able to move to the next step, that of specifying actors and their relationships.

*3.3. Actors Formalization.* "An actor specifies a role played by a user or any other system that interacts with the subject" [3]. According to the UML standard, an actor is defined as a classifier. Actually the given set *CLASSIFIER* is used to define the set of all unique identifiers throughout the specification. The variable *ACTOR* is introduced as a classifier to specify the set of actors' identifiers.

$$ACTOR: \mathbb{P}\ CLASSIFIER$$

The actor can be human user, some internal application or may be some external application. The actor type is introduced as an enumerated set representing two types of actors.

$$ActorTYPE ::= HUMAN \mid nonHUMAN$$

For actors, there are mainly two types of relationships: (1) a relationship between actor and use case and (2) a relationship between actors that is nothing other than generalization.

The relationship between actor and use case is basically represented as a Z relation that relates an actor to a use case. The relation *associationAU* is used to record which use case can be used by the actor, whereas the function *relatedUsecases* returns the set of use cases that are associated to the actor.

$associationAU: ACTOR \leftrightarrow Usecase$
$relatedUsecases: ACTOR \rightarrow \mathbb{P}\ Usecase$

---

$\forall\ a: ACTOR \bullet relatedUsecases\ a = \{\ uc: Usecase \mid (a, uc) \in associationAU\ \}$

Actor generalization is used to factor out and reuse similarities between actors. This relationship shows that one actor inherits the role and properties of another actor. To represent generalization, we chose to add a state variable to declaration part of the schema *Actor* as shown below. This state variable is called *parents* and specified as a set of actors. The generalization relationship also implies that the descendant actor can use all the use cases that have been defined for its ancestor. This is clearly stated in the predicate part.

*Actor*

$$
\begin{aligned}
&self: ACTOR\\
&actorName: NAME\\
&actorType: ActorTYPE\\
&relatedUC: \mathbb{P}\ Usecase\\
&parents: \mathbb{P}\ ACTOR\\
&inheritedUC: \mathbb{P}\ Usecase\\
&\hline
&relatedUC = relatedUsecases\ self \cup inheritedUC
\end{aligned}
$$

The function *getActorFromACTOR* takes an actor identifier and return the corresponding actor. This function will be used later in the use case diagram formalization.

$$
\begin{aligned}
&getActorFromACTOR: ACTOR \rightarrow Actor\\
&\hline
&\forall\ a: ACTOR; ac: Actor \mid ac.self = a \bullet getActorFromACTOR\ a = ac
\end{aligned}
$$

Similarly, we define the function *getUsecaseFromUCASE*.

$$
\begin{aligned}
&getUsecaseFromUCASE: UCASE \rightarrow Usecase\\
&\hline
&\forall\ UC: UCASE; uc: Usecase \mid uc.self = UC \bullet getUsecaseFromUCASE\ UC = uc
\end{aligned}
$$

*3.4. Use Case Diagram Formalization.* Based on the formalization of use cases, actors and their relationships, the concept of use case diagram can be fully defined. As shown in the following schema, a use case diagram has a name and it is composed of a set of actors and a set of use cases.

We also specify the scenario of execution of use cases in the case of extension. We propose the function *UCexecution* which takes a sequence of use cases fragments and extension points and returns a boolean type. This function indicates whether or not the extension will be executed.

*useCaseDiagram*

$$
\begin{aligned}
&name: NAME\\
&actors: \mathbb{P}\ Actor\\
&usecases: \mathbb{P}\ Usecase\\
&UCexecution: \text{seq}\ (FRAGMENT \times ExtensionPoint \times FRAGMENT) \rightarrow BOOL\\
&\hline
&\forall\ a: Actor; ap: \mathbb{P}\ ACTOR \mid a \in actors \land ap = a.parents\\
&\quad \bullet\ a.inheritedUC = \cup \{\ x: ap \bullet (getActorFromACTOR\ x).relatedUC\ \}\\
&\forall\ a1, a2: Actor \bullet a1 \in actors \land a2 \in actors \Leftrightarrow a1.self \neq a2.self\\
&\forall\ u1, u2: Usecase \bullet u1 \in usecases \land u2 \in usecases \Leftrightarrow u1.self \neq u2.self\\
&\forall\ u1, u2: Usecase \mid u1 \in usecases \land u2 \in usecases\\
&\quad \bullet\ u1.self \in u2.includedUC \Rightarrow u2.self \notin u1.includedUC\\
&\forall\ u1, u2: Usecase; f1, f2: FRAGMENT; e: ExtensionPoint\\
&\quad \mid u1 \in usecases\\
&\qquad \land\ u2 \in usecases\\
&\qquad \land\ getUsecaseFromUCASE^{\sim}\ u2 \in u1.extendingUC\\
&\qquad \land\ \langle f1 \rangle \subseteq u1.Fragments\\
&\qquad \land\ \langle f2 \rangle \subseteq u2.Fragments\\
&\qquad \land\ \langle e \rangle \subseteq u1.extensionPoints\\
&\quad \bullet\ \textbf{if}\ e.constraint = \{True\} \lor e.constraint = \varnothing\\
&\qquad \textbf{then}\ UCexecution\ \langle (f1, e, f2) \rangle = True\\
&\qquad \textbf{else}\ UCexecution\ \langle (f1, e, f2) \rangle = False
\end{aligned}
$$

The first predicate states that, for each actor, the inherited use cases are nothing other than related use cases of its parents. It is also important to note that use cases and actors identifiers are unique. The fourth predicate states that a use case cannot include use cases that directly or indirectly include it. The last predicate means that an extension occurs only if the constraint is true or if the extension is unconditional.

Now we concentrate on how to use the formal specification of use case diagram defined in this section to checking the multiview consistency. In the following section, we discuss the consistency between use case diagram, class diagram and sequence diagram.

## 4. Multiview Consistency Checking

Having discussed how to construct a formal model of use case diagram using Z notation, the final section of this paper addresses ways of checking consistency in multiview context. Multiview modeling is a methodology for describing different aspects of the system in terms of different views or models. This methodology presents several challenges, particularly those related to consistency, which could potentially put the integrity of the system at risk. "In general, a system is consistent when there is logical coherence (i.e., no contradictions) between its parts" [23]. For these reasons, it is essential that all views are shown to be consistent and coherent between them.

Once the formal model of use case diagrams has been defined, it is useful to state and prove theorems between different views. This helps to verify the system and check for errors. In this paper, we focus on consistency of three different models of a system, comprising class diagram, sequence diagram, and use case diagram. Note that class diagram and sequence diagram formal models and their constraints are, respectively, defined in Z notation in [6, 7].

*4.1. Consistency between Use Case Diagram and Sequence Diagram.* To formalize the notion of consistency between use case diagram and sequence diagram we consider a Z theorem called *consistencyUsecaseAndSequenceDiagram.* This theorem checks whether

(i) each use case name corresponds to a frame of sequence diagram;

(ii) actors in use case diagram are defined as objects in sequence diagram.

**theorem** *consistencyUsecaseAndSequenceDiagram*
$\forall$ *ud: useCaseDiagram; uc: Usecase; a: Actor*
| $uc \in ud.usecases \land a \in ud.actors \land uc \in a.relatedUC$
• $\exists$ *sd: SequenceDiagram* • $sd.Frame = uc.name \land a.self \in sd.Objects$

Here, for any use case diagram *ud*, a use case *uc* and an actor *a* such that *uc* and *a* belong, respectively, to *ud* use cases and actors and *uc* is in the set of use cases related to *a;* then there exists a sequence diagram *sd* such that its frame corresponds to the name of *uc* and the identifier of *a* is in the set of *sd* objects.

If the theorem is proved to be true, then every constraint expressed before could be satisfied and consequently this would guarantee the system always remains in consistent state.

*4.2. Consistency between Use Case Diagram and Class Diagram.* Let us now define the notion of consistency between use case diagram and class diagram.

First, given an instance of class diagram and a use case diagram, we define a Z theorem called *consistencyUsecaseAndClassDiagram*. The theorem claims that the two views are consistent, if we assume that

(i) Actors correspond to defined objects of classes in class diagram.

(ii) Use cases in use case diagram are assigned to operations of classes in class diagram.

We define the function *op* which, given a use case, returns the corresponding operation.

| $op: Usecase \rightarrow OP$

**theorem** *consistencyUsecaseAndClassDiagram*
$\forall$ *ud: useCaseDiagram; a: Actor; uc: Usecase*
| $a \in ud.actors \land uc \in ud.usecases$
• $\exists$ *class: CLASS; operation: OP*
• $a.self \in ObjectsOfClass\ class \land (uc, operation) \in op$

Here, for any use case diagram *ud*, a use case *uc* and an actor *a* such that *uc* and *a* belong, respectively, to *ud* use cases and actors; then there exists a class named *class* such that *a* identifier is in the set of objects of *class* and *uc* corresponds to one of its operations.

The proposed theorem ensures that the specification satisfy the constraints presented before. If the constraints are satisfied then the specification is consistent. The theorem should be proved to be true.

For further information on class diagrams and sequence diagrams formalizations, the reader is referred to [6, 7].

## 5. Conclusions and Future Work

"The specification of requirements is a key activity for achieving the goals of any software project and it has long been established and recognized by researchers and practitioners" [24]. In this paper, we have considered a formal specification of the major concepts of use case diagram. The objectives of our work are providing concise and unambiguous specification of use case diagrams using Z notation and consistency checking by means of Z theorems. Formal specification of use case diagrams has been done in the past using different languages but we believe that the expressiveness and the mathematical basis of Z notation provide a richer framework for such formalism. The main motivation behind our research was the fact that the use of formal specification can greatly enhance the quality of the produced software as well as to contribute significantly to cost savings in the software development process.

One important field for further research is to find ways to handle larger views in order to detect more inconsistencies. We also plan to extend our prototype to include use case diagram formalization. We reiterate that our prototype automatically translates a subset of UML diagrams into a Z formal specification to uncover most of the UML inconsistencies published to date. Furthermore, we will make our approach more practical using some industrial applications examples. All the presented specifications were syntactically checked and proven to be true using the Z/EVES system [25] which is a tool for analysing Z specifications supporting type checking and theorems proving.

## Data Availability

No data were used to support this study.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## References

[1] W. S. Humphrey, "The Software Quality Challenge," *CROSSTALK: The Journal of Defense Software Engineering*, pp. 4–9, 2008.

[2] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, Reading, UK, 1st edition, 1999.

[3] "OMG Unified Modeling Language Superstructure 2.4.1," 2011, http://www.omg.org/spec/UML/2.4.1/Superstructure.

[4] J. M. Spivey, *The Z Notation: A Reference Manual*, Oriel College, Oxford, England, 2nd edition, 1998.

[5] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*, Prentice-Hall, Upper Saddle River, NJ, USA, 1996.

[6] K. El Miloudi, Y. El Amrani, and A. Ettouhami, "An automated translation of UML class diagrams into a formal specification to detect UML inconsistencies," in *Proceedings of the 6th International Conf. on Software Engineering Advances*, pp. 432–438, Barcelona, Spain, 2011.

[7] K. El Miloudi, Y. El Amrani, and A. Ettouhami, "Using Z formal specification for ensuring consistency in multi-view modeling," *Journal of Theoretical and Applied Information Technology*, vol. 57, no. 3, pp. 407–411, 2013.

[8] K. El Miloudi and A. Ettouhami, "A multi-view approach for formalizing UML state machine diagrams using Z notation," *WSEAS Transactions on Computers*, vol. 14, pp. 72–78, 2015.

[9] A. Bollin, "Metrics for quantifying evolutionary changes in Z specifications," *Journal of Software: Evolution and Process*, vol. 25, no. 9, pp. 1027–1059, 2013.

[10] A. Hall, "Realising the Benefits of Formal Methods," in *Formal Methods and Software Engineering*, vol. 3785 of *Lecture Notes in Computer Science*, pp. 1–4, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[11] J. Oliveira, L. Ribeiro, E. Cota et al., "Use case analysis based on formal methods: an empirical study," in *Recent trends in algebraic development techniques*, vol. 9463 of *Lecture Notes in Comput. Sci.*, pp. 110–130, Springer International Publishing, Switzerland, 2015.

[12] D. Sinnig, P. Chalin, and F. Khendek, "Use case and task models," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 3, pp. 1–31, 2013.

[13] G. Butler, P. Grogono, and F. Khendek, "A Z specification of use cases," in *Proceedings of the 5th Asia-Pacific Software Engineering Conference*, pp. 94–101, Taipei, Taiwan, 1998.

[14] F. V. Barajas, "A formal model for a requirements engineering tool," in *Proceedings of the 1st. Alloy Workshop*, pp. 63–70, Portland, Ore, USA.

[15] P. Scandurra, A. Arnoldi, T. Yue, and M. Dolci, "Functional requirements validation by transforming use case models into Abstract State Machines," in *Proceedings of the 27th Symposium On Applied Computing*, pp. 1063–1068, ACM Press, New York, NY, USA, March 2012.

[16] W. Shen and S. Liu, "Formalization, Testing and Execution of a Use Case Diagram," in *Formal Methods and Software Engineering*, vol. 2885 of *Lecture Notes in Computer Science*, pp. 68–85, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[17] R. Murali, A. Ireland, and G. Grov, "A Rigorous Approach to Combining Use Case Modelling and Accident Scenarios," in *NASA Formal Methods*, vol. 9058 of *Lecture Notes in Computer Science*, pp. 263–278, Springer International Publishing, Cham, 2015.

[18] A. M. Mostafa, M. A. Ismail, H. EL-Bolok, and E. M. Saad, "Toward a Formalization of UML2.0 Metamodel using Z Specifications," in *Proceedings of the Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, pp. 694–701, Qingdao, China, July 2007.

[19] T. Yue, L. C. Briand, and Y. Labiche, "aToucan," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 3, pp. 1–52, 2015.

[20] T. Yue, L. C. Briand, and Y. Labiche, "Automatically deriving UML sequence diagrams from use cases," Tech. Rep. TR-SCE-10-03, Carleton University, Canada, 2010.

[21] M. Giese and R. Heldal, "From informal to formal specifications in UML," in *Proceedings of the 7th Int. Conf. «UML» 2004—The Unified Modeling Language. Modeling Languages and Applications 2004*, vol. 3273 of *Lecture Notes in Computer Science*, pp. 197–211, Springer Berlin Heidelberg, Heidelberg, Germany, 2004.

[22] N. Ibrahim, R. Ibrahim, M. Z. Saringat, R. Mansor, and T. Herawan, "Use case driven based rules in ensuring consistency of UML model," *AWERProcedia Information Technology and Computer Science*, vol. 1, pp. 1485–1491, 2012.

[23] M. Alférez, R. E. Lopez-Herrejon, A. Moreira, V. Amaral, and A. Egyed, "Consistency checking in early software product line specifications - The VCC approach," *Journal of Universal Computer Science*, vol. 20, no. 5, pp. 640–665, 2015.

[24] R. P. Oliveira, E. Insfran, S. Abrahao et al., "A Feature-Driven Requirements Engineering Approach for Software Product Lines," in *Proceedings of the 2013 VII Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pp. 1–10, Brasília, DF, Brazil, September 2013.

[25] I. Meisels, "Software Manual for Windows Z/EVES Version 2.3," Tech. Rep. TR-97-5505-04h, Ora, Canada, 2004.