*Journal of Engineering Sciences, Assiut University, Vol. 34, No. 1, pp. 173-188, January 2006*

# TOWARD A FORMALIZATION OF USE CASE DIAGRAM USING Z SPECIFICATIONS

***Ahmed M. Mostafa*** [1] ; ***Manal A. Ismail*** [2] ; ***El Sayed M. Saad*** [3] ***and Hatem EL-Bolok*** [4]
*Electronics & Communication Department, Faculty of Engineering, Helwan University, Cairo, Egypt*

[1] *engahmed_youssef@ yahoo.com*
[2] *mshoman@helwan. edu.eg*
[3] *elsayedmos@ hotmail.com*
[4] *hbolok@mailer.eun.eg*

*(Received September 24, 2005 Accepted November 27, 2005)*

*The Unified Modeling Language (UML) is an evolutionary step in the development of Object-Oriented analysis and design (OOA&D) methods that appeared in the late '80s and early '90s. UML has become a widely adopted standard in the software development industry. Various attempts have been made to formally define the syntax and semantics of the UML notations, and to represent its models in a formal notation. The purpose of these attempts is to allow UML models to be rigorously checked, and to allow formal analysis of the modeled systems. Use Case Diagram is one of the diagrams supported by UML which describe the functional requirements of the system under development, helping to identify the complete set of user requirements. This paper aims to present a tentative approach to provide the Use Case Diagram with formal semantics using Z specification language.*

***KEYWORDS:*** *Unified Modeling Language (UML), Use Case Diagram, Formalization, Z specification language.*

## 1. INTRODUCTION

The Unified Modeling Language (UML) [1] is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems.

Unfortunately, UML has many limitations that preclude rigorous (or sound) development. UML models are imprecise and cannot be formally analyzed in the UML context. This brings the following consequences [2]:

1. UML models result in ambiguous descriptions of software systems.
2. UML models cannot be checked for consistency, which means that one may produce unsatisfiable models for which no implementation may possibly exist.

3. There are no means for checking whether certain desired system properties hold in a UML model.

Formal specification languages (FSLs), on the other hand, yield precise descriptions of software systems that are amenable to formal analysis. However, these languages require substantial expertise from developers.
There are some of the ways in which the UML could benefit from formalization [3]; these ways are:

- **Clarity:**

To act as a reference - if at any point, there is confusion over the exact meaning of a particular UML component; reference can be made to the formal description to verify its semantics.

- **Equivalence and Consistency:**

To provide an unambiguous basis from which to compare and contrast the UML with other techniques and notations, and for ensuring consistency between its different components.

- **Extendibility:**

To enable the soundness of any extensions to the UML to be verified (as encouraged by the UML authors).

- **Refinement:**

To allow correctness of design steps in the UML to be verified and precisely documented. In particular, it should enable design patterns to be checked for correctness. Once checked, a particular pattern can be used again and again without having to re-check it.

- **Proof:**

To allow justified proofs and checks of important properties of a system described in the UML, for example safety properties.

The use case model can serve as a means of communication between the different stakeholders in a project. It is used in planning the project and is updated and used during the project [4]. Developing use cases is one of the first steps in the object oriented approach using UML to capture the required functionality [5]. Use cases, while very useful, are nevertheless informal descriptions suffering from the problems of inconsistencies, ambiguities etc. Even though use cases are the starting point in UML based software engineering, there is very little by way of formalization [6].

This paper presents an initial attempt to provide a suitable formal model for the use Case Model using Z Specifications. Section 2 presents the Use Case Model in an informal way to determine its underlying semantic foundation. A brief definition of Z specifications is presented in section 3. Section 4, presents a formal description of the basic Use Case Model concepts using Z specifications, and finally Section 5 is a conclusion.

## 2. AN INFORMAL DESCRIPTION OF USE CASE MODEL

The elements in the Use Cases model are primarily used to define the behavior of an entity, like a system or a subsystem, without specifying its internal structure. The key elements in this model are *Use Case* and *Actor*.

**Figure 1** shows some of the use cases for a financial trading system as an example of a use case diagram [7]. An informal description of the use case model will be given through this example. Figure 1 shows that the use case diagram consists of four basic concepts. These concepts are *Actor*, *Actor Relationships*, *Use Case*, and *Use Case Relationships*.

## 2.1. Actor

An *actor* defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates. An Actor may also have a set of Interfaces; each describing how other elements may communicate with the Actor.

Actors carry out use cases. A single actor may perform many use cases; conversely, a use case may have several actors performing it. There are four actors in figure 1; *Trading Manager*, *Trader*, *Salesperson*, and *Accounting System*. Actors don't need to be human; an actor can also be an external system (i.e. *Accounting System*).

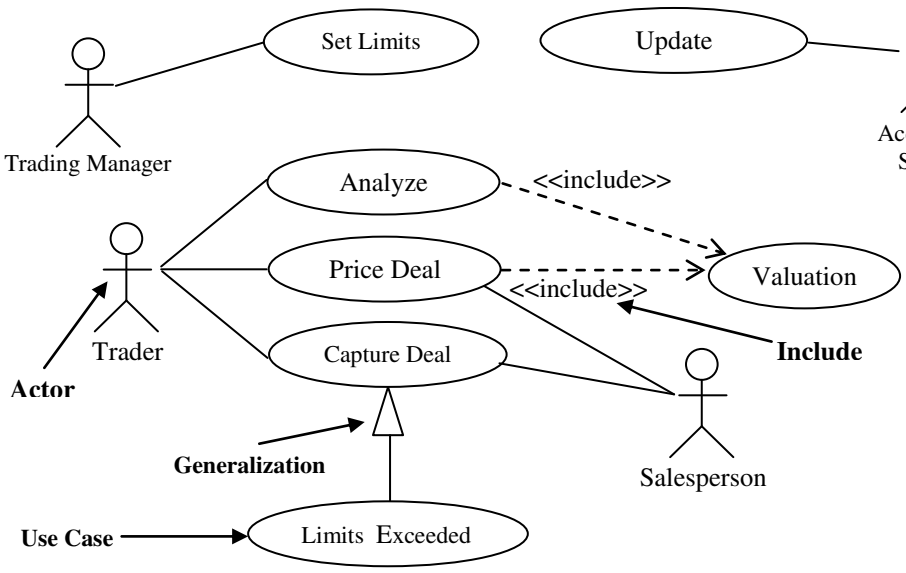


**Figure 1:** Use Case Diagram.

## 2.2. Actor Relationships

There is one standard relationship among actors (*Generalization*) and one between actors and use cases (*Association*).

• *Generalization* – An actor may have generalization relationships to other actors. This means that the child actor will be able to play the same roles as the parent actor, that is, communicate with the same set of use cases, as the parent actor. **Figure 2** shows an example of actor generalization, where the *Trading Manager* actor will be able to play the same roles as the *Manager* actor.

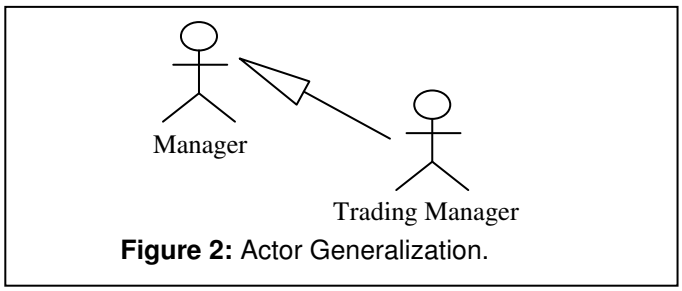


**Figure 2:** Actor Generalization.

• *Association* – The participation of an actor in a use case; that is, instances of the actor and instances of the use case communicate with each other. This is the only relationship between actors and use cases. For example, the association between *Accounting System* actor and *Update Accounts* use case.

## 2.3. Use Case

A *use case* is a set of scenarios tied together by a common user goal. As an example, consider the Buy a Product use case with the successful purchase and the authorization failure as two of the use case's scenarios. There are seven use cases in **figure 1**; *Set Limits*, *Update Accounts*, *Analyze Risk*, *Price Deal*, *Capture Deal*, *Limits Exceeded*, and *Valuation*.

## 2.4. Use Case Relationships

In addition to the links among actors and use cases (*Association*), several kinds of relationships between use cases (*Include*, *Generalization*, *Extend*) can be shown.

• *Include* – An *include* relationship means that a chunk of behavior is similar across more than one use case and it is not wanted to keep copying the description of that behavior. For instance, both *Analyze Risk* and *Price Deal* require valuing the deal. Use *include* to avoid repetition in two or more separate use cases.

• *Generalization* – A *generalization* relationship between use cases implies that the child use case contains all the attributes, sequences of behavior, and extension points defined in the parent use case, and participates in all relationships of the parent use case. The child use case may also define new behavior sequences, as well as add additional behavior into and specialize existing behavior of the inherited ones. In our example, the basic use case is *Capture Deal*; this is the case in which all goes smoothly. Things can upset the smooth capture of a deal, however. One is when a limit is exceeded – for instance, the maximum amount the trading organization has established for a particular customer. Here we carry out an alternative to the usual use case. Use *generalization* to describe a variation on normal behavior casually.

• *Extend* – An *extend* relationship is similar to generalization but with more rules to it. The extending use case may add behavior to the base use case, but this time the base use case must declare certain "extension points" and the extending use case may add additional behavior only at one or more of these extension points. **Figure 3** shows an example of the extend relationship, here the customer is already known to the system

as a regular customer and the system will display the current shipping, pricing, and billing information. Use *extend* to describe a variation on normal behavior using the more controlled form, declaring extension points in the base use case.

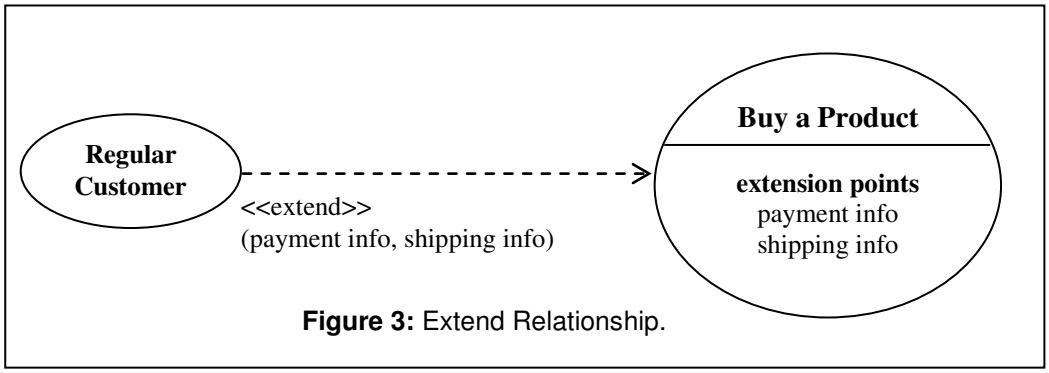


**Figure 3:** Extend Relationship.

## 3. Z SPECIFICATIONS

A formal specification can serve as a single, reliable reference point for all those concerned with the system [8]. Formal Methods can be grouped under three general categories [9]:

***Set based formalisms*:** Use mathematical notation to describe in a precise way the properties which an information system must have. These are good for describing systems in an "object oriented" way. Provides a high level view of a system that can be refined as specification proceeds. Example of this is: Z specification language.

***Logic based formalisms*:** This is a wide field which includes specification languages and property languages. Examples of this are classical logic, predicate logic, modal logics, temporal logics, and theorem provers.

***Behavior based formalisms*:** Systems are described as states and transitions between states. A natural way of describing a system for a programmer. Examples in this area include labeled transition systems, and Petri Nets.

For the UML formalization, Z specification language is most appropriate because it is a mature, expressive, and abstract language that is well supported by tools [10].

In Z, specification can be decomposed into small pieces called schemas. A schema consists of two parts, the declaration part which declare some variables and the predicate part which gives a set of relationships between the variables in the declaration part.

Schemas can be written in one of two forms:

- **Vertical form:**

Schema-Name
Declaration
---
Predicate; : : : ; Predicate

❖ **Horizontal form:**

Schema-Name ≙ Schema-Exp

These forms introduce a new schema name. If the predicate part in the vertical form is absent, the default is true. In the horizontal form, the Schema-Exp which comes after the definition sign (≙) can be any number of schema names connected by logical operators.

The schema Aleph is an example of schemas:

Aleph
x,y: ℤ
---
x < y

Which declare two variables x and y as integers, and state that x must be smaller than y.

A schema binding is the assignment of values to its variables declared in the declaration part. For example, the binding ⟨x ⇛ 3, y ⇛ 5⟩ satisfies the condition x < y. The expression Aleph.x = 3 selects only the x component of the schema Aleph.

Now, a summary of Z notations which are used in the formalization of the use case diagram is given in the next subsections.

## 3.1. Set Operators

- **Power set**

  If S is a set, ℙ S is the set of all subsets of S.

  $\mathbb{P}_{\square}$ represents non-empty set.

- **Set difference**

  The members of S \ T are those objects which are members of S but not of T.

## 3.2. Relations

- **Binary Relations**

  If S and T are sets, then S ↔ T is the set of binary relations between S and T.

  S ↔ T = ℙ (S × T).

- **Domain and Range**

  dom, ran - represents the domain and range of a relation.

- **Partial and Total Functions**

  If X and Y are sets, X ⇸ Y is the set of partial functions from X to Y.
  These are relations which relate each member x of X to at most one member of Y. This member of Y, if it exists, is written f(x). The set X → Y is the set of total functions from X to Y. These are partial functions whose domain is the whole of X; they relate each member of X to exactly one member of Y.

### 3.3. Logical operators

$\neg$ Negation
$\wedge$ Conjunction
$\vee$ Disjunction
$\Rightarrow$ Implication (note: not $\rightarrow$)
$\Leftrightarrow$ Equivalence (note: not $\leftrightarrow$)

### 3.4. Numbers and Finiteness

$\mathbb{N}$ Is the set of natural numbers {0; 1; 2; ……}.

$\mathbb{Z}$ Is the set of integers {…… ;-2;-1; 0; 1; 2; ….. }.

$\mathbb{F}$ Members of a set can be counted by a natural number.

# There is a unique natural number which counts the members of the set S without repetition, and this is the size #S of S.

### 3.5. Sequences

- **Finite sequences (seq)**

  seq x is the set of finite sequences over x. For example, the sequence $\langle x_1, x_2, \ldots\ldots, x_n \rangle$ can be written as the set $\{1 \mapsto x_1, 2 \mapsto x_2, \ldots\ldots, n \mapsto x_n\}$, where:
  $x \mapsto y$ is a graphic way of expressing the ordered pair (x,y).
- **Non-empty Finite sequences ($seq_1$)**
  Is the set of all finite sequences over x except the empty sequence.
- **Disjointness and Partitions**
  Disjointness between the two sets A, B means they do not intersect and the sets A, B partition the set C means that C is the union of A and B and that the two sets A, B do not intersect.

### 3.6. Quantification

$Q\ x_1 : S_1; \ldots\ldots ; x_n : S_n \mid p \bullet q$
Where Q is one of $\forall$ (for all), $\exists$ (there exist). Meaning:
$\forall\ x_1 : S_1; \ldots\ldots ; x_n : S_n\ (p \Rightarrow q)$
Whatever the value taken by the variables $x_1$ to $x_n$ which make p true, the predicate q will be true as well.
$\exists\ x_1 : S_1; \ldots\ldots ; x_n : S_n\ (p \wedge q)$
There is at least one way of giving values to the variables $x_1$ to $x_n$ so that both predicates p and q true.

## 4. FORMALIZATION OF THE BASIC USE CASE MODEL

The basic *Use Case model* consists of [1]:

- Actors.
- Use Cases.

- Actor Relationships.
- Use Case Relationships.

The result of the formalization will be a Z schema for the basic *Use Case Model* that includes four schemas that correspond to the above concepts.

BasicUseCaseModel

Actors
Usecases
ActorRelationships
UseCaseRelationships

## 4.1. Actors Schema:

It is assumed that there is a given sets:
[ActorName, Role]
From which the names of all actors and their roles can be drawn.

Actors

human, non_human: $\mathbb{F}$ ActorName
actorRoles: $\mathbb{F}$ Role
aRoles: ActorName $\rightarrow \mathbb{F}$ Role

$\langle$human, non_human$\rangle$ Partition ActorName

dom aRoles = human $\cup$ non_human

Y (ran aRoles) = actorRoles

The constraint in the predicate part of the schema state that human and non_human actors can not have the same name.

## 4.2. Use cases Schema:

It is assumed that there is a given sets:
[UsecaseName, Actions, ExtensionPoints]
From which the names of all use cases, actions, and extension points can be drawn.

Usecases

ucases: $\mathbb{F}$ UsecaseName
scenario: $\text{seq}_1$ Actions
usecase: $\mathbb{P}$ scenario
expoints: $\mathbb{F}$ ExtensionPoints
extension: UsecaseName $\nrightarrow \mathbb{F}$ ExtensionPoints

dom extension = ucases
Y (ran extension) = expoints
$\forall$ u: ucases; $\forall$ x: extension(u) | #( extension(u)) > 1 $\bullet$ (extension(u)\{x}) $\cap$ {x}= $\varnothing$

The constraint in the predicate part of the schema state that the names of the extension points must be unique in the same use case.

## 4.3. Actor Relationships Schema:

It is assumed that there is a given sets:
[AssociationEndName, AssociationName]
From which the names of all association ends and associations can be drawn.

AssociationEnd

Actors
Usecases
owner: $\mathbb{F}$ AssociationEndName

disjoint $\langle$(human $\cup$ non_human), ucases$\rangle$

owner $\subseteq$ (human $\cup$ non_human $\cup$ ucases)

The constraint of the schema state that the use cases's names and actors's names can not be the same, and the owner of the association end is an actor or a use case.

Association

AssociatioEnd
associations: $\mathbb{F}$AssociationName
linking: AssociationName $\rightarrow \mathbb{F}$ AssociatioEnd

$\forall$ a: associations; $e_1$,$e_2$: AssociationEnd | #(linking(a)) = 2 $\wedge$ $e_1$= (linking(a)(1)) $\wedge$
$e_2$ = (linking(a)(2)) $\bullet$ (($e_1$.owner $\in$ (human $\cup$ non_human) $\Rightarrow$ ($e_2$.owner $\in$ ucases)) $\vee$
(($e_2$.owner $\in$ (human $\cup$ non_human) $\Rightarrow$ ($e_1$.owner $\in$ ucases))

The constraint of the schema is just a way of saying that associations only allowed between use cases and actors and these associations are binary associations.

Actor Generalization

Actors
aparent, achild: $\mathbb{F}$ Actors
ageneralization: aparent $\leftrightarrow$ achild

$\forall$ x,y,z: human | (x,y) $\in$ ageneralization $\wedge$ (y,z) $\in$ ageneralization $\bullet$ (x,z) $\in$ ageneralization
$\forall$ x,y,z: non-human | (x,y) $\in$ ageneralization $\wedge$(y,z) $\in$ ageneralization $\bullet$ (x,z) $\in$ ageneralization
$\forall$ p: aparent; $\forall$ c: achild | (p,c) $\in$ ageneralization $\bullet$ (p.aRoles $\subseteq$ c.aRoles $\wedge$(c,p) $\notin$ ageneralization)

The constraint of the schema says that the child actor inherits all roles of its parent and its parent can not inherit any thing from it.
The schema *ActorRelatioShips* is the conjunction of the above schemas:

*ActorRelatioShips* $\triangleq$ *Association* $\wedge$ *ActorGeneralization*

## 4.4. Use case Relationships Schema:

Usecase Generalization

Usecases
uparent, uchild: $\mathbb{F}$ Ucases
ugeneralization: uparent $\leftrightarrow$ uchild

$\forall$ x,y,z: ucases | (x,y) $\in$ ugeneralization $\wedge$ (y,z) $\in$ ugeneralization $\bullet$ (x,z) $\in$ ugeneralization
$\forall$ p: uparent; $\forall$ c: uchild | (p,c) $\in$ ugeneralization $\bullet$
((p.usecase $\subseteq$ c.usecase) $\wedge$ (extension(p) $\subseteq$ extension(c)) $\wedge$ ((c,p) $\notin$ ugeneralization))

The constraint of the schema says that the child use case inherits all use cases and extension points of its parent and its parent can not inherits any thing from it.

UsecaseExtend

Usecases
extending, base: $\mathbb{F}$ Usecases
extendingpoints: $seq_1$ ExtensionPoints
condition, isextended:Bool
extend: extending $\leftrightarrow$ base

condition = t $\Rightarrow$ isextended = t
ran(extendingpoints) $\subseteq$ extension(base)

The constraint of this schema is that the referenced extension points must be included in the set of extension points of the base use case.

UsecaseInclude

Usecases
included, base: $\mathbb{F}$ Usecases
include: base $\leftrightarrow$ included

$\forall$ x,y,z: ucases | (x,y) $\in$ include $\wedge$ (y,z) $\in$ include $\bullet$ (x,z) $\in$ include
$\forall$ b: base; $\forall$ i: included | (b,i) $\in$ include $\bullet$ (( i.usecase $\subseteq$ b.usecase) $\wedge$ ((i,b) $\notin$ include))

The constraint here is that the set of scenarios in the included use case is a subset of the base usecase's scenarios and the included use case can not include its base use case.

The schema *UseCaseRelationships* is the conjunction of the above schemas:

$$\begin{aligned} \textit{UseCaseRelationships} \triangleq\ & \textit{Association} \wedge \\ & \textit{UsecaseGeneralization} \wedge \\ & \textit{UsecaseExtend} \wedge \\ & \textit{UsecaseInclude} \end{aligned}$$

Now, consider the example given in section 2 (**figures 1, 2,** and **3**) to be represented as Z specifications using the previous schemas. The use case diagram has the following sets (*human*, *non-human*, *ucases*, *expoints*, *owner*, *associations*, *aparent*, *achild*, *ageneralization*, *uparent*, *uchild*, *ugeneralization*, *extend*, *include*):

- human
  {Manager, Trading Manager, Trader, Salesperson}
- non-human
  {Accounting System}

- ucases
  {Set Limits, Analyze Risk, Price Deal, Capture Deal, Valuation, Limits Exceeded, Update
  Accounts, Regular Customer, Buy a Product}
- expoints
  {payment info, shipping info}
- owner
  {Trading Manager, Trader, Salesperson, Accounting System, Set Limits, Analyze Risk, Price Deal, Capture Deal, Update Accounts}
- associations
  {Trading Manager_Set Limits, Accounting System_Update Accounts, Trader_Analyze Risk, Trader_Price Deal, Trader_Capture Deal, Salesperson_Price Deal, Salesperson _Capture Deal}
- aparent
  {Manager}
- achild
  {Trading Manager}
- ageneralization
  {(Manager, Trading Manager)}
- uparent
  {Capture Deal}
- uchild
  {Limits Exceeded}
- ugeneralization
  {(Capture Deal, Limits Exceeded)}
- extend
  {(Regular Customer, Buy a Product)}
- include
  {(Analyze Risk, Valuation), (Price Deal, Valuation)}

The specification schemas are as follows:

**Actors**

human, non_human: $\mathbb{F}$ ActorName

⟨human, non_human⟩ Partition {Manager, Trading Manager, Trader, Salesperson, Accounting System }

**Usecases**

ucases: $\mathbb{F}$ UsecaseName
expoints: $\mathbb{F}$ ExtensionPoints
extension: UsecaseName ⇸ $\mathbb{F}$ ExtensionPoints
dom extension = {Buy a Product}
extension (Buy a Product) = {payment info, shipping info}
Buy a Product: ucases; payment info: extension(Buy a Product) | #(extension (Buy a Product))>1 ●
(extension (BuyProduct) \ { payment info }) ∩ { payment info } = ∅
Buy a Product: ucases; shipping info: extension(Buy a Product) | #(extension (Buy a Product))>1 ●
(extension (BuyProduct) \ { shipping info }) ∩ { shipping info } = ∅

Association End

Actors
Usecases
owner: $\mathbb{F}$ AssociationEndName

disjoint ⟨{Manager,Trading Manager, Trader, Salesperson, Accounting System}, {Set Limits, Analyze Risk, Price Deal, Capture Deal, Valuation, Limits Exceeded, Update Accounts, Regular Customer, Buy a Product}⟩

{Trading Manager, Trader, Salesperson, Accounting System, Set Limits, Analyze Risk, Price Deal, Capture Deal, Update Accounts} ⊆ { Manager, Trading Manager, Trader, Salesperson, Accounting System, Set Limits, Analyze Risk, Price Deal, Capture Deal, Valuation, Limits Exceeded, Update Accounts, Regular Customer, Buy a Product}

Association

Associatio End
associations: $\mathbb{F}$ AssociationName
linking: AssociationName → $\mathbb{F}$ AssociatioEnd

Trading Manager_Set Limits: associations; Trading Manager, Set Limits: AssociationEnd |
# (linking ( Trading Manager_Set Limits)) = 2 ∧
Trading Manager = (linking ( Trading Manager_Set Limits) (1)) ∧
Set Limits = (linking ( Trading Manager_Set Limits) (2)) ●
(Trading Manager ∈ human) ⇒ (Set Limits ∈ ucases)

Accounting System_Update Accounts: associations;
Accounting System, Update Accounts: AssociationEnd |
#(linking(Accounting System_Update Accounts)) = 2 ∧
Accounting System = (linking(Accounting System_Update Accounts) (1)) ∧
Update Accounts = (linking(Accounting System_Update Accounts) (2)) ●
(Accounting System ∈ non-human) ⇒ (Update Accounts ∈ ucases)

Trader_Analyze Risk: associations; Trader, Analyze Risk: AssociationEnd |
#(linking(Trader_Analyze Risk)) = 2 ∧Trader = (linking(Trader_Analyze Risk) (1)) ∧
Analyze Risk = (linking ( Trader_Analyze Risk) (2)) ● (Trader ∈ human) ⇒ (Analyze Risk ∈ ucases)

Trader_Price Deal: associations; Trader, Price Deal: AssociationEnd |
#(linking(Trader_ Price Deal)) = 2 ∧Trader = (linking(Trader_ Price Deal) (1)) ∧
Price Deal = (linking(Trader_ Price Deal) (2)) ● (Trader ∈ human) ⇒ (Price Deal ∈ ucases)

Trader_Capture Deal: associations; Trader, Capture Deal: AssociationEnd |
#(linking(Trader_ Capture Deal)) = 2 ∧Trader = (linking(Trader_ Capture Deal) (1)) ∧
Capture Deal = ( linking ( Trader_ Capture Deal ) (2)) ● ( Trader ∈ human) ⇒ ( Capture Deal ∈ ucases)

Salesperson_Price Deal: associations; Salesperson, Price Deal: AssociationEnd |
# (linking ( Salesperson _ Price Deal )) = 2 ∧ Salesperson = ( linking ( Salesperson _ Price Deal) (1)) ∧
Price Deal = (linking(Salesperson _ Price Deal) (2)) ●
(Salesperson ∈ human) ⇒ (Price Deal ∈ ucases)

Salesperson_ Capture Deal: associations; Salesperson, Capture Deal: AssociationEnd |
# (linking(Salesperson _ Capture Deal)) = 2 ∧Salesperson = (linking(Salesperson _ Capture Deal) (1)) ∧
Price Deal = (linking(Salesperson _ Capture Deal) (2)) ●
(Salesperson ∈ human) ⇒ (Capture Deal ∈ ucases))

ActorGeneralization

Actors
aparent, achild: $\mathbb{F}$ Actors
ageneralization: aparent $\leftrightarrow$ achild

Manager: aparent; Trading Manager: achild | (Manager, Trading Manager) $\in$ ageneralization $\bullet$ (Trading Manager, Manager) $\notin$ ageneralization

Usecase Generalization

Usecases
uparent, uchild: $\mathbb{F}$ Ucases
ugeneralization: uparent $\leftrightarrow$ uchild

Capture Deal: uparent; Limits Exceeded: uchild | (Capture Deal, Limits Exceeded) $\in$ ugeneralization $\bullet$ (Limits Exceeded, Capture Deal) $\notin$ ugeneralization

Usecase Extend

Usecases
extending, base: $\mathbb{F}$ Usecases
extendingpoints: $\text{seq}_1$ ExtensionPoints
extend: extending $\leftrightarrow$ base

Regular Customer: extending; Buy a Product: base; extending points = $\langle$payment info, shipping info$\rangle$; ran(extendingpoints) ={payment info, shipping info};
extension(Buy a Product) = {payment info, shipping info} $\bullet$
ran(extendingpoints) $\subseteq$ extension(Buy a Product)

Usecase Include

Usecases
included, base: $\mathbb{F}$ Usecases
include: base $\leftrightarrow$ included

Analyze Risk: base; Valuation: included | (Analyze Risk, Valuation) $\in$ include $\bullet$ (Valuation, Analyze Risk) $\notin$ include

Price Deal: base; Valuation: included | (Price Deal, Valuation) $\in$ include $\bullet$ (Valuation, Price Deal) $\notin$ include

Consider the extend relationship shown in **figure 4**; the Z specification of this relationship can not be generated.

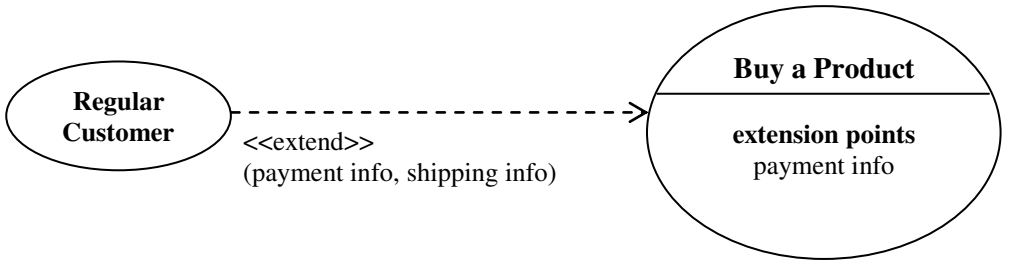


**Figure 4:** Extend Relationship with error.

This error occurred because the second predicate (ran (extendingpoints) $\subseteq$ extension (base)) in the *UseCaseExtend* schema is invalid. From **figure 4**, it can be seen that ran(extendingpoints) is {payment info, shipping info}; extension(Buy a Product) is {payment info}.

In addition to this error, the Z specification presented in section 3 can check other types of errors. These errors are presented in table 1 which consists of two columns, the first is the UML requirements and the second is the Z specification constraints to satisfy them.

## 5. CONCLUSION

Although mature object-oriented analysis modeling techniques are widely used for software specification, their expressiveness and rich set of intuitive constructs are not used for the modeling of complex systems. This is mainly due to their lack of support for rigorous analysis [11]. This paper has presented a tentative approach for formalizing the Unified Modeling Language (UML). As an initial step in this direction, a representation of the use case diagram using Z specification has been proposed. It has been shown that the UML has many benefits from formalization; these are clarity, equivalence and consistency, extendibility, refinement, and proof. All the above work is very important in developing a more precise understanding of emerging software development techniques.

By concentrating on the use case diagram of the UML, it is aimed to develop a more understandable and manageable description of the language.

**Table 1.** UML Requirements against Z Specification Constraints.

| | UML Requirements | Z Specification Constraints |
|---|---|---|
| 1 | Human and non_human actors can not have the same name. | ⟨human, non_human⟩ Partition ActorName |
| 2 | The names of the extension points must be unique in the same use case. | ∀ u: ucases; ∀ x: extension(u) \| #( extension(u)) > 1 ● (extension(u) \ {x}) ∩ {x} = ∅ |
| 3 | The use cases's names and actors's names can not be the same, and the owner of the association end is an actor or a use case. | disjoint ⟨(human ∪ non_human), ucases⟩ owner ⊆ (human ∪ non_human ∪ ucases) |
| 4 | Associations only allowed between use cases and actors and these associations are binary associations. | ∀ a: associations; $e_1$,$e_2$: AssociationEnd \| #(linking(a)) = 2 ∧ $e_1$= (linking(a) (1)) ∧ $e_2$ = (linking(a) (2)) ● (($e_1$.owner ∈ (human ∪ non_human) ⇒ ($e_2$.owner ∈ ucases)) ∨ (($e_2$.owner ∈ (human ∪ non_human) ⇒ ($e_1$.owner ∈ ucases)) |
| 5 | The child actor inherits all roles of its parent and its parent can not inherit any thing from it. | ∀ p: aparent; ∀ c: achild \| (p,c) ∈ ageneralization ● (c,p) ∉ ageneralization |
| 6 | The child use case inherits all use cases and extension points of its parent and its parent can not inherits any thing from it. | ∀ p: uparent; ∀ c: uchild \| (p,c) ∈ ugeneralization ● (c,p) ∉ ugeneralization |
| 7 | The referenced extension points in an extend relationship must be included in the set of extension points of the base use case. | ran(extendingpoints) ⊆ extension (base) |
| 8 | The included use case can not include its base use case. | ∀ b: base; ∀ i: included \| (b,i) ∈ include ● (i,b) ∉ include |

## REFERENCES

[1] Object Management Group (OMG), "OMG Unified Modeling Language Specification", version 1.5, URL "http://www.omg.org/uml", March 2003.

[2] N. Amálio, S. Stepney, and F. Polack, "*Formal Proof from UML Models*", ICFEM 2004, Seattle, USA, pp 418-433, Springer, 2004

[3] A.S. Evans and A.N. Clark, "*Foundations of the Unified Modeling Language*", 2nd Northern Formal Methods Workshop, electronic Workshops in Computing, Springer-Verlag, 1998.

[4] B. Anda and M. Jørgensen, "Understanding Use Case Models", International Conference on Software Engineering, pp 94-102, June 5, 2000, Limerick, Ireland.
[5] G. Booch, J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1999.
[6] P. Krishnan, "A Framework for Analyses of Use Case Descriptions", URL "http://www.it. bond.edu.au/publications", 2003.
[7] M. FOWLER and K. SCOTT, "The UML Distilled", second edition. Addison-Wesley, 2000.
[8] J. Michael Spivey, "The Z Notation: A Reference Manual", Prentice Hall, Englewood Cliffs, NJ, Second edition, 2001.
[9] T. Tynjälä, "Case Study 1: Formal Methods – Introduction", URL "http://www.tcs.hut.fi/ Studies/T-79.232", February 2, 2005.
[10] R. France, A. Evans, K. Lano, B. Rumpe, "The UML as a Formal Modeling Notation", the pUMLgroup, URL "http://www.cs.york.ac.uk/puml/papers", 1998.
[11] J.-M. Bruel and R.B. France, "Transforming UML Models to Formal Specifications",
*Proceedings of the Int. Conf. on Object Oriented Programming Systems Language and Applications (OOPSLA'98) Vancouver, Canada*, 18-22 October 1998.

## نحو التمثيل الشكلي لنموذج حالة الاستخدام بواسطة مواصفات زد

لغة النمذجة الموحدة هي خطوة متقدمة في تطوير طرق التحليل و التصميم باستخدام الأشياء والتي ظهرت في أواخر الثمانينات و أوائل التسعينات. لقد أصبحت لغة النمذجة الموحدة مقياس واسع الاختيار في صناعة تطوير البرمجيات. محاولات عديدة تمت للتمثيل الشكلي لتركيبات ومعاني رموز لغة النمذجة الموحدة. الغرض من هذه المحاولات هو التحقق الدقيق من نماذج لغة النمذجة الموحدة والسماح بالتحليل الشكلي للأنظمة المصممة. الرسم التخطيطي لحالة الاستخدام هو احد الرسومات التخطيطية التي تدعمها لغة النمذجة الموحدة و الذي يصف الاحتياجات الوظيفية للنظام المطور، و ذلك للمساعدة علي التعرف علي الاحتياجات الكاملة للمستخدم. هذا البحث يهدف لتقديم طريقة مبدئية نحو التمثيل الشكلي لنموذج حالة الاستخدام بواسطة مواصفات زد.