Abstract

The UML 1.1, as released in September 1997 represents an improved version of the previous document by giving revised definitions of many modelling elements and including well-formedness rules, defining its four layered architecture and organisation by package, and including an Object Constraint Language Specification used within well-formedness rules. In this paper we analyse and critically assess some modelling solutions with observations regarding types, interfaces and classifiers, and problems of associations semantics and notation. Well formedness rules, their format and role are also analysed. Use case modelling is assessed through their role in eliciting user’s requirements, and dealing with the functionality of the system. Some examples show the potential danger of using use-cases as a replacement for functional decomposition. This could result in some important abstractions in the system not being revealed and not seeing use cases as a representation of collective functionality where their components act as collaborations between objects.

Introduction

The UML as represented in the documentation set [1] delivered in September 1997 is a valuable upgrading of the previous document [2] published earlier in the year. The UML is defined as a standard modelling language1 for specifying, visualising, constructing and documenting the artifacts of software systems, as well as for business modelling and other non-software systems2. As a modelling language the UML is aimed to be applicable in the context of different processes where the UML authors promote a use-case driven, architecture centric, and iterative and incremental development process3 like Rational Objectory Process. The documentation set itself consists of UML Summary that justifies the motivation to define the UML, its goals and scope, and summary of changes between versions 1.0 and 1.1; UML Semantics that defines the semantics and syntax of the UML, its layered architecture and organisation by package; UML Notation Guide that defines notation and provides supporting examples in the form of graphic syntax for expressing the semantics described by the UML metamodel; UML Extension for Business Modelling and UML Extension for Objectory Process for Software Engineering that include process- and domain-specific extensions to the UML in terms of its extension mechanisms and process-specific diagram icons; Object Constraint Language Specification (OCL) used by the UML and defined separately4. This set of documents is successfully targeting the authors’ intended audience where familiarity with OO modelling issues is required in order to understand and apply the UML modelling constructs and its notation. However, the UML Semantics document uses improved definitions of the language itself - compared with the previous version [2] - through three different views for each modelling element: abstract syntax, well-formedness rules and semantics. This gives a satisfactorily formal definition of the UML and easiness of understanding the majority of modelling elements.

1 UML Summary, v1.1, page 3
2 UML Summary, v1.1, page 1
3 UML Summary, v1.1, page 5
4 UML Summary, v1.1, page 1
In this paper we analyse the UML through its semantics and notation, consulting documents from [1]. In the first chapter we look at the UML structure. The second chapter critically assesses some modelling solutions and changes from the UML version 1.0 with observations regarding types/interfaces and classifiers, problems of different aspects of associations (under unification of relationship semantics) and problems of notations that were discussed in previous works [3]. The third chapter deals with the issue of clearly specified rules that are supposed to be followed in order to claim that the UML itself is followed. Some observations to well-formedness rules are made with the discussion of automating all rules in CASE tools that supports the UML as it was analysed in [4]. The fourth chapter covers business modelling stereotypes that promote use-case driven procedures. We assess how use-case modelling is seen as an effective way of communicating and eliciting user’s requirements. Observations are made on the role of actor and initiations of the activities covered by use-cases, the problem of granularity and abstractions of use-cases using “uses” and “extends”, and accompanying use-case generalisation. Finally we compare the data flow diagramming technique as a part of functional decomposition with the role of use-cases: there is a danger of using use-cases as a replacement for functional decomposition when not seeing them as a representation of collective functionality where their components act as collaborations between objects.

1. The UML Structure

The UML specification comprises two complimentary documents/parts: UML Semantics, with the abstract syntax of the language itself and UML Notation, a visual representation of the UML semantics that represents the mapping of the graphic notation to the underlying semantics⁵ (both are consulted in this paper). The architecture of the UML is based on a four layer metamodel structure: user objects, model, metamodel and meta-metamodel⁶ where UML metamodel defines the complete semantics for object modelling using UML. The UML structure within the metamodel layer covers logical packages: Foundation and Behavioural Elements and General Mechanisms where each of them are decomposed into sub-packages: Foundation Elements consists of Core and Auxiliary Elements, Extension Mechanisms and Data Types, Behavioural Elements cover Common Behaviour, Collaborations, Use Cases and State Machines and General Mechanisms comprise the Model Management Package. The structure of the language itself gives us the description of the metamodel through three different views (for each sub-package): a syntax which can be found under Abstract Syntax headings, then Well-formedness Rules that describe the static semantics (except ‘multiplicity’ and ‘ordered’ requirements that are part of the syntax), and Semantics that uses a natural language in order to explain the meanings of the constructs (concrete metaclasses). We can find also in some sub-packages Standard Elements where stereotypes of the metaclasses defined previously are listed and Notes subsection that gives us rationales for some metamodelling decisions. The semantics of the UML is applicable to structural (static) and behavioural (dynamic) object models. Structural models cover classes and their attributes, their interfaces and relations. Behavioural models deal with methods, interactions, collaborations and state histories. This includes a range of diagrams: class diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, package diagram, activity diagram and deployment diagram.

2. Assessment of some Modelling Solutions

(a) Classifiers and Types

UML version 1.0 introduced Types to denote the essence of abstraction, having PrimitiveType (e.g. enumeration, numbers...), Class and UseCase as subclasses of the Type. By this separation of Types and Classes, a Class instance is seen as a realisation of a Type instance, Class is supposed to be implemented but Type may not be

---

⁵ UML Semantics, v1.1, page 1
⁶ UML Semantics, v1.1, page 6
⁷ UML Semantics, v1.1, page 11
implemented at all. This Type/Class dichotomy provoked various discussions/confusions [5],[3] and required this problem to be addressed in following versions of the UML. This resulted in the Types/Class dichotomy being almost removed and introducing a Classifier as a superclass of Class, DataType and Interface. A Type is still present within the UML 1.1 in the Glossary of the Semantics document and defined as a “...stereotype of class used to specify a domain of instances (objects) together with operations applicable to the objects”\(^8\). The Notation document distinguishes between Type and Implementation Class through one example\(^9\) and stating that Class can be specialised by stereotypes into Types and Implementation Classes, although they can also be left undifferentiated.\(^10\).

The Classifier is introduced in the Semantics’ document as an abstract meta-class which describes behavioural and structural features and may participate in association. Some features of classifier and association may be required to participate in collaborations through classifier roles \(^11\). Furthermore, behavioural elements Actor and UseCase are also subclasses of a Classifier where Actor may communicate at realisation level with Classifiers that take part in realisation of UseCases. However a UseCase, as a subclass of a Classifier, may have a set of Operations and Attributes specifying the sequences of actions performed by an instance of the UseCase\(^12\). Classifier may also participate in Activity Models which describe a state model of an activity process involving one or more Classifier through ClassifierinState (instances of a given classifier for a particular state)\(^13\).

All the issues above may raise many questions: Are the problems resulted from Type/Class dichotomy as discussed in [3],[6] now solved? Do Classifiers replace Types?

The way Types and ImplementationClass are represented in the UML 1.1 should remove some problems, particularly if we have freedom not to differentiate between them. At different levels of abstraction a Class could be seen as a Type or a proper implementation element called ImplementationClass. They are both specialisations of a Class which in turn could be successfully used within the analysis or design stage as appropriate. This view makes a Type/Class dichotomy extended to analysis/design dichotomy not obvious. It also allows us to interpret a Class freely as the logical modelling element within any modelling stage, splitting it into a Type or Implementation Class due to e.g. implementation language requirements or because of building some other components that might require this separation.

A Classifier is a modelling element that comes in various specific forms as specified earlier. As an abstract metamodelling element it is not instantiable, it shows sharing constructs and reifying key constructs like GeneralisableElement or ModelElement. This is a good compromise if abstractions are used when defining meta-modelling elements. This is also a better wording: the term ‘classifier’ does not overload the word ‘type’ and still carries enough semantics in order to employ abstractions.

(b) Association, Aggregation and Composition

The UML1.1 does not include any significant change when describing associations. The definition of aggregation remains ambiguous, the composition is again defined as a strong ownership and all attributes regarding association ends (it was association role in version 1.0.) remained the same (except the additional one targetScope that specifies whether the targets are ordinary Instances or are Classifiers). This raises more questions than discussed in [3]:

\(^8\) UML Semantics, v1.1, page 159  
\(^9\) UML Notation, v1.1, page 36, figure 12  
\(^10\) UML Notation, v1.1, page 35  
\(^11\) UML Semantics, v1.1, page 86  
\(^12\) UML Semantics, v1.1, page 90  
\(^13\) UML Semantics, v1.1, page 123
(a) contrasting the semantics of association, aggregation and composition relationship,
(b) implementing composition (e.g. lifetime dependency!) when discussing structural concepts,
(c) using changeable attribute in order to change the way you model an association throughout a life-cycle,
(d) using isNavigable attribute, as a core concept used for developing object diagram, in order to specify implementation and specification issues that are not likely to be part of any conceptual design.

The choice of all these modelling constructs is justified by the authors’ intention to preserve many aspects of relationships that have already been recognised and used in Booch [7] and OMT [8]; e.g. ‘uses’ as defined in Booch [7] is mapped in UML association which isNavigable attribute has value True, composite aggregation is meant to be akin to aggregation-by-value and non-composite aggregation is akin to aggregation-by-reference. The UML does not reduce the relationship semantics in order to clearly distinguish between conceptual design and implementation. The concurrent analysis and design activities are answers to this problem when we are reluctant to take some design decisions very early in the life cycle.

However, the UML1.1. offers better presentation of association semantics through different attributes of AssociationEnd by giving explicitly all choices that are applicable to each attribute. E.g. ChangableKind could have frozen value when no links may be added after the creation of source object, or addOnly value when a link, once created, may not be removed before at least one participating object is destroyed\textsuperscript{14}.

\textbf{(c) Notation}

The UML 1.1 keeps the same philosophy of including textual elements as a part of graphical presentation for the majority of its modelling constructs and diagrams, as in the previous version 1.0 [3]. The number of attributes attached to some modelling elements, which in addition can be in textual form, does not contribute to desirable readability of many UML diagrams which results in heavy dependency on an appropriate software tool support in order to keep the modelling element’s labels and adornments meaningful during the development process (particularly in later stages). The good choice of symbols that comprise notation should improve communication between developers, and contribute to the methodology’s compactness and ease of comprehensiveness. It is difficult to see how the UML notation can do both, except if we eliminate ‘unnecessary’ symbols and simplify the notation as it was represented in [9].

\textbf{3. The Role of Well-formedness Rules}

The UML is represented as a language (not a method!) that can support any methodology for OO systems development by being process free and not including any process/strategy definitions. However, the UML covers some aspects that are expected to be addressed by any OO methodology, such as:

(a) a set of concepts and guidelines that define all modelling elements,
(b) notation acceptable for users and developers,
(c) a set of rules that represent the philosophy of the methodology that are supposed to be followed in order to claim that the methodology itself is followed,
(d) fully described deliverables and their employment within different models,

as it is described in [4]. The existence of clearly specified rules (c) has proved to be very valuable from different points of view. Generally accepted rules for any OO analysis and design activity could be summarised and contribute significantly to better applicability in the development process (or even to more successful

\textsuperscript{14} UML Semantics, v1.1, page 18
delivery/teaching within academic curricula). Furthermore, these rules are valuable methodology checks that can be automated in a CASE Tool that supports a particular methodology, and they represent a basis of any evaluation instrument which determines what role a particular CASE tool plays as a UML ‘companion’ [10]. In previous versions of the UML all rules had to be extracted mainly from semantics and notation documents [11],[4]. In the current UML version we can see them better defined in the form of well-formedness rules as one of three different views of the UML metamodel. However, there are some obstacles and inconsistencies in their specification.

The well-formedness rules are part of static semantics defined in the UML which are to be fulfilled by well-formed constructs from dynamic semantics (described as abstract syntax under the headings Semantics)

15. Some of the well-formedness rules like ‘multiplicity’ and ‘ordered’ constraints on relationships are defined in diagrams which are part of abstract syntax, showing all modelling constructs and their relationships. All these rules are defined as a set of invariants of an instance of a metaclass that are to be satisfied for the construct to be meaningful

16. The rules specify all constraints over attributes and associations of each modelling construct in informal explanation and OCL expression.

(A) However, some of the rules (not related to multiplicity and ordered constraint) can be found within a short informal description of a construct’s abstract syntax and not represented within a section of well-formedness rules. An example is the definition of an Association construct that includes the sentence “Each tuple value may appear at most once”

17, or an AssociationEnd construct which ChangeableKind attribute ‘frozen’ specifies that ‘No links may be added after the creation of the source object’

18 which could be very important rules to follow, but can not be found within any well-formedness rules specified later within the same chapter.

(B) Some other rules for e.g. Association construct are explicitly specified in informal language like: “An Association has at least two AssociationEnds”, or “the same Classifier may be connected to more than one AssociationEnds in the same Association” and are directly mapped from the diagram (Figure 6: Core Package-Relationships diagram) which is justified by the multiplicity constraint being represented within an abstract syntax. These rules are consequently not repeated within the well-formedness rules section (which is expected), but the first rule is repeated within ‘connection’ (which is the association section of Association construct’s abstract syntax).

(C) Furthermore, the only attribute of Association construct named ‘name’ is represented in abstract syntax through the rule: “The name of Association which, in combination with its associated Classifier, must be unique within an enclosing namespace (usually a Package)” which gives a clear guidelines for name uniqueness so desirable in any modelling and is expected also to be found within well-formedness rules. It is not found within an Association construct’s well-formedness rules (which is expected) but within the well-formedness rules of a Namespace construct

19. This somehow contradicts the authors’ decision to allow to express all current semantics of a construct element in its superclass which could be, according to the figure 6 Core Package-Relationships diagram a ModelElement for an Association construct where we can impose a constraint on its ‘name’ attribute and successfully cover all possible types of ModelElement name’s uniqueness. A very experience modeller can trace the name uniqueness rule easily after reading the Namespace abstract syntax where “a Namespace is a ModelElement and can own other ModelElement”

20 and after reading well-formedness rule number 4 for Association, but this can be a confusing issue for any attempt to quickly map abstract syntax issues represented in different diagrams and informal language, with corresponding well-formedness rules.

15 UML Semantics, v1.1, page 10
16 UML Semantics, v1.1, page 11
17 UML Semantics, v1.1, page 17
18 UML Semantics, v1.1, page 18
19 UML Semantics, v1.1, page 33
20 UML Semantics, v1.1, page 26
The problem described in section (A) above is the result of a strict division of the UML syntax into static (well-formedness rules) and dynamic (abstract syntax) which immediately allows some ‘static rules’ (like multiplicity and ordered constraints) to be declared within ‘dynamic section’ and some dynamic issues represented in form of rules that are definitely as important as well-formedness rules (and can not find their place within them). The problem described in section (B) is the result of inconsistency in abstract syntax between the short informal description of modelling constructs and short explanation of modelling construct’s attributes and opposite role names of associations (connected to the modelling construct itself). The problem analysed in section (C) above is a consequence of weak structure and connection between static and dynamic semantics of the UML constructs which evidently points out the role and importance of the well-formedness rules.

The solution to the problems above should include a section on all rules (as in (c)) to be fully and carefully stated separately from abstract syntax and referred from abstract syntax whenever necessary. Rules should include well-formedness rules and rules regarding all other issues (dynamics?) listed in hierarchical (generalisation) order as it is represented in corresponding diagrams (graphic notation). This would be of significant value to any modeler and CASE tools developer in order to use/support the UML and would improve the readability of the whole UML Semantics document.

4 Use Cases Modelling

All traditional/structured methodologies are based on generally accepted principles of developing systems using functional, top-down approach. The context level clearly specifies the scope of the system whose boundary is to be maintained throughout different levels of the model. The system’s environment is represented in external entities that are not part of, but may influence the system’s processes by sending inputs to, and accepting outputs from the system. The system is functionally decomposed into subsystems using data flow diagrams that allow us to explode all processes to the lower levels, until we reach functional primitives which can be described using process specification techniques like decision tables/trees. Throughout analysis/design steps the emphasis is on the strict differentiation between logical and physical aspects of the system which does not allow any implementation details to be incorporated early in the development cycle.

However, this is not suitable for implementing aspects of OO paradigm in spite of having some sound solutions that are supposed to be abandoned due to different modelling principles in OO environment. It will be discussed in the following chapters how use cases accommodate some of these concepts, by contrasting them with user’s requirements, data flow diagrams (DFD) and looking at issues like logical/physical and use case granularity.

4.1. Use Cases versus Users Requirements

One of the most problematic tasks in the systems development process is understanding and capturing user’s requirements. Requirements Engineering (RE) has emerged lately and reinforced the need to elicit, specify and analyse user’s requirements. The UML does not dedicate any of its modelling elements specifically to RE, but we can see use cases contributing to eliciting user’s requirements.

The issue of capturing user’s requirements within OO modelling has not been addressed so often in existing methodologies and the introduction of Use Cases in [12] and Task Scripts in [13] are more than welcome. Searching requirements for nouns in order to detect all relevant abstractions in the system (and contribute towards RE), as described in OMT [8], proved to be unsuccessful if the system requirements were not well specified. An introduction of use cases in previous UML versions resulted in some practitioners combining them with their scenarios as a way of discovering analysis classes and their behaviour. Furthermore, if you apply the OMT’s noun discovery procedure on existing use cases you could have a better success in finding viable objects. In the
UML 0.9 addendum, the mapping of use cases and classes was introduced, but UML 1.1 explicitly defines class stereotypes that use cases are mapped onto. ‘Entity’, ‘control’ and ‘boundary’ (or ‘interface’ in OOSE [12]) classes are treated as predefined stereotypes of classes: each use case can contain a ‘control scenario’ encapsulated within an object, it can manipulate data represented in an ‘entity’ object and must interact with the user in some form of ‘boundary’ object. This bridges the gap between use cases and the object model (and its sequence diagram in particular) to be developed. It is called a Robustness Analysis as described in [14] which makes the gap between issues like WHAT (the system is supposed to do) and HOW (is it going to be accomplished) smaller.

Many different kinds of requirements exist: performance, test and functionality requirements. All of them are very often in written form with sentences that use ‘must’ and ‘shall’ words. We should expect that use cases can accommodate the majority of them, particularly functional requirements, and that some other requirements can map into methods, attributes or even create a new group of classes that are not otherwise associated. Sequence diagrams could include some timing requirements. This shows that there is no 1:1 mapping between requirements and use cases. Use cases successfully bridge the gap between user’s perception of the system and modelling solutions, i.e. connecting a user’s view of the system with potential objects and they are not initially designed to elicit user’s requirements. They describe how the system is going to be used through different roles of actors. However they could help us to consider the reusability mechanism within RE activities by identifying common requirements that could be extended and re-used. Both use cases and formal requirements are very important techniques that complement each other and can not be used instead of each other (if we do not want the user to pay a price). The developers could take advantage of robustness analysis which gives the opportunity to reconsider some use cases and discover all necessary objects, particularly ones that could be missed out in any other approach (e.g. solely data-oriented or process oriented approach).

4.2. Use Cases versus Data Flow Diagrams

DFDs have been widely used in many structured methodologies as a technique for analysis and partially for eliciting users requirements (through analysis of the current system). Many developers have been very confident in their use when determining the scope of the system i.e. system boundaries and their role in the functional decomposition which is the backbone of all traditional process driven methodologies. In an OO environment the collective functionality of the system is represented through use cases which are also designed for eliciting user’s requirements but they model the external behaviour of the system (user’s view) leaving internal behaviour for an object model developed after robustness analysis. The differences between DFDs and use cases are easy to spot, but similarities between them could lead us towards unwanted solutions when doing an OO analysis.

A functionality of the system can not be avoided whichever approach you use. Objects exist with their functionality that makes a contribution to the overall functionality of the system. However, there will always be a danger of strictly applying the functional decomposition we learned with DFDs, to use cases. This may result in different functional partitions containing objects in common, i.e. a functional view of the system scattering objects across many partitions (not always desirable). Furthermore, functional decomposition ends with functional primitive processes that are implemented as a programming routine (function) and it will be very difficult to attach them to a particular class. In addition, under which circumstances shall we allow classes to encapsulate functions only? The solution to this problem requires two aspects to be monitored:

(a) A decomposition of use cases should strictly follow OO architectural concepts (not functional concepts). Use cases should always be used as an external view of any package/subpackage being analysed/developed with clear interface and control (packaging mechanism could be used for use case decomposition with boundary objects detected early). The role of an actor helps. They are always external to the component you are trying to understand/decompose.
(b) Representing systems internals should always start with robustness analysis as suggested in [14] which reveals all boundary and control objects mentioned above and continues with discovering all objects and collaborations between them (use cases are very often seen as an object discovery technique - they are analysed through robustness analysis).

The UML is a language not a process and we do not expect to read any of the recommendations mentioned above in the document. However, if some modelling element definitions are enriched with examples in the Notation, or in the textual format of the Semantics document, there should be more guidelines available for use cases. The UML does not give any suggestions on how to deal with the functionality of the system and does not give examples of transitions from use cases to object model (including the role of robustness analysis). This is of a vital interest if UML modelling elements are to support business processes and engineering: objects should be defined if necessary across the functional (vertical) boundaries of the system where they can be mapped to business processes.

The following scenario pictures the functionality of the system and the role of use cases. We can consider a simplified example from [15] of Payroll System for all employees in the company regardless of the way and the time of month they are paid. Hourly employed workers submit their daily time cards with the date and number of hours worked. They are paid an hourly rate (which is recorded in their employee record) and they receive their paycheque every Friday. Salaried employees are paid a flat salary (amount kept in their records) on the last working day in the month. If they are also to be paid a commission, they need to submit sales receipts which record the date and the amount of the sale. Their commission rate is kept in their employee record and they are paid the commission every second Friday. Some employees may belong to the Union and their weekly dues rate (kept in their employee record) must be deducted from their pay. Service charges against individual union members are submitted on a weekly basis. This also must be deducted from the appropriate employee’s next pay.

The use of Data Flow Diagrams clearly reveals the scope of the system by defining Employee and Union as external entities and revealing some processes that can be easily detected in the text as represented in Figure 1. However, any attempt to define use cases along the functional paths from Figure 1 will result in some underlying abstractions, so important for OO development not being detected, and some abstractions to be employed across both functional paths not being found. These underlying abstraction are important. They will enable us to create software modules that can not be changed (source code is frozen) but are open to different extensions in order to suit new requirements [16]. E.g. different methods of calculating salary and different methods of payments can not be abstracted straightforwardly through DFDs although we could have some hints on a possible hierarchy that creates an Employee superclass from Hourly- paid, Salaried and Commissioned Employees. Furthermore, DFD from Figure 1 does not show any possible association between Employees and Union Members (both are external entities) which contributes in the class model. In spite of dealing with the overall functionality of the system, use cases should not strictly follow any functional path. They should be constructed in a way that shows how external actors from the domain model interact with the whole system, and how they “use” the system to accomplish a particular task. Figure 2 lists some use cases that lead towards the class model represented in Figure 3. The list of use cases does not follow any of the functional paths defined in Figure 1. They answer questions like “what are the activities that the user (actor) can undertake with and expect from the system”. An actor can initiate many activities, from updating employee details including adding a new employee, to running the whole payroll system (and expecting appropriate pay to be calculated). All these use cases must be further expended with appropriate scenarios. The robustness analysis reveals some interface and entity objects like Sales Receipt, Timecard, Monthly Payment, that can participate in many use cases and control objects like Payment Scheduler which has to participate in the control flow that handles the payment calculation.
4.3. Logical versus Physical

To maintain a distinction between logical and physical views of the system is regarded as a sound modelling principle. Indeed, use cases are supposed to be a ‘what model’ (giving an external view of the system) as opposed to the object model which is a ‘how model’[14]. This is regulated through their definition and rules that restrict showing a communication between instances of use cases (leaving this for the object model) and restrict showing parallel courses of events in the use case model. However, an ambiguous sentence in [14] which says that use cases are the way of ensuring that anyone who uses the system can agree on how the model will look\(^{21}\) can cause confusion and misinterpretation. How to solve the problem of e.g. describing HOW a user interacts with the system with use cases; or detecting some important objects before you have understood the context in which these domain objects exist and interact (a situation often exploited in prototyping)? In both cases you are prone to jump too early into a design field and ruin the barrier between logical and physical views. When developing use cases descriptions with too much HOW information, such as user interface screens, you are drafting how users would interact with the system, not how the business process is achieved at the logical level. None of these problems are addressed in use case texts apart from rules that advise you to restrict use cases by determining only ‘WHAT the system has to do’ and not include any ‘HOW the system will work’.

4.4. Use Cases and <<uses>> / <<extends>> Stereotypes

In order to describe associations between use cases, UML defines the generalisations stereotypes\(^{22}\) called <<uses>> and <<extends>>. An extends relationship defines an extension of an existing use case with some additional behaviour defined in another use case. A condition that allows an extension is specified, thus extends represents a ‘conditional logic’. A uses relationship expresses commonalties between use cases, i.e. includes the notion of ‘common behaviour’ across use cases (common logic of an abstract use case?).

There are several problems associated with the definitions above:

(a) In both documents, Notation and Semantics, uses and extends represent stereotypes of generalisation relationship. They are expected to represent inheritance or aggregation (as it has been suggested in [17]). However, this directly contradicts the sentence which says that “Behaviour specified by several extenders of a single target use case may occur within a single use case instance”\(^{23}\) and does not appear to follow generalisation/specialisation hierarchy semantics at all. Would it be more appropriate to see extends and uses as stereotyped dependency? Furthermore, it is difficult to see how one part of a process (defined through the behaviour) could be a specialised form of the more general process.

(b) Without having more explanations on both generalisation stereotypes, developers are left to their own conclusions and assume that extends might represent optionality (the presence of ‘conditional logic’) where extended use case might be logically complete without extending one. Uses could represent ‘part-of’ semantics and might initially be used throughout modelling. It could be replaced with extends as soon as optionality arises if we find examining conditions for optionalities extremely important in a particular model.

(c) If we allocate extends for exception handling, how do we solve the problem of exception on common behaviour? The solution might be in interpreting extending use cases as being different scenarios of one use case. However, this will definitely have impact on re-usability of use cases where we might interpret an extended use case as an abstraction.

\(^{22}\) UML Semantics, page 90
\(^{23}\) UML Notation, v1.1, page 78
5. Conclusions

In this paper the UML 1.1 is viewed, and some modelling constructs chosen to be assessed, from three different perspectives.

Firstly, the authors’ intention has been to analyse the basic modelling constructs like classes, their abstractions and associations between some of them after delivering the UML within an academic curriculum. The class/type dichotomy extended to analysis/design dichotomy and the complex nature of associations as discussed in [3] and [6] provoked different interpretations among UML practitioners and required more clarification when teaching them. It appears that the UML 1.1 does not solve many of these problems particularly when discussing some implementation issues within a conceptual design (e.g. aggregation/composition), or not clearly specifying the role of types/classes within logical/physical specification. However, the introduction of classifiers in the UML 1.1 allows us to see a class as the logical modelling element and to simplify the role/need of abstractions, which is equally important for inexperienced UML practitioners as for students without substantial background in OO programming languages.

Secondly, we look in this paper at the role of use cases in the UML and how they accommodate some aspects of system functionality (e.g. functional decomposition) so prevalent in traditional process driven methodologies. In spite of not dedicating any of the modelling constructs to eliciting users’ requirements the UML defines through Actors/Use Cases some abstractions in the form of entity/control/boundary classes that could lead us towards discovering classes employed across functionally different paths of the system. Use cases can carry functionality of the system as whole, but their scenarios do not necessarily match the paths of functional decomposition. This answers many questions of applying some sound modelling principles of structured methodologies within an OO paradigm, and points out the danger of misinterpreting use cases (often the case when students are taught process driven methodologies initially or practitioners used functional decomposition extensively in the past).

Thirdly, we acknowledge the significance of OO CASE tool support in using the UML and importance of a clearly specified set of rules that represents the philosophy of the UML in order to claim that the UML itself is being followed. This version 1.1 explicitly defines well-formedness rules as one of the three different views of the UML metamodel. Rules specify all constraints over attributes and associations of each modelling construct in informal language and OCL expression. This is a very important element of any modelling language/method and represents an improvement from the UML 1.0 where all these rules were supposed to be extracted manually. However, there are some inconsistencies in their specification due to an unfortunate division of syntax to static/dynamic and weak structure/connection between static and dynamic semantics of the UML constructs. The well formedness rules should be allocated separate section within the UML document set, they should be referenced from abstract syntax whenever necessary and represented in hierarchical order. This would improve the readability of the document and help students/practitioners easily to detect violations of the rules prescribed by the UML.

However, in this paper we do not discuss and assess so many other issues that are important for the complete analysis of the UML modelling constructs, and the role of the UML in the systems development process. This includes some UML limitations like: the lack of method for modelling user interface of the system (can we use a state diagram to compensate the problem?); the lack of well-defined guidelines for checking consistency and completeness among the diagrams; the problem of similarity/distinction and the use of sequence and collaboration diagrams. All these issues should be addressed in the future work.
REFERENCES:


