An Automated Translation of UML Class Diagrams into a Formal Specification to Detect UML Inconsistencies

Khadija El Miloudi Younès El Amrani Aziz Ettouhami
Laboratoire Conception et Systèmes FSR
University MohamedV-Agdal
BP 1014 RP Rabat. Morocco
{elmiloudi,elamrani,touhami}@fsr.ac.ma

Abstract— **In view of the informal semantic of UML, there is a high risk of introducing ambiguities and contradictions in the modelled software. A considerable amount of literature has been published on UML inconsistencies. These studies have demonstrated the absence of any rule in UML to prevent such inconsistencies from being introduced in UML designs. This article describes a systematic translation of UML Class Diagrams into a formal specification to uncover most of the UML inconsistencies published to date. Examples of inconsistent UML class diagrams presented in previous research studies were used to validate the approach. The formal model obtained from UML class diagrams helped to uncover inconsistencies without any further proof. In order to relieve the user from writing a much rigorous and precise formalism, a tool that automatically generates the formal model from the UML class diagram was developed.**

Keywords- Z; UML; UML inconsistencies; Formal Specification; Software Model Checking.

I. INTRODUCTION

There are numerous off-the-shelf software proposing to automatically translate UML Class Diagrams into several implementations. Nonetheless, there are very few translations into a formal notation to detect UML inconsistencies. Therefore, UML inconsistencies are insidiously injected into any generated implementation, when not removed. According to a definition provided in [7], inconsistency "denotes any situation in which a set of descriptions does not obey some relationship that should hold between them". This paper will use this definition to identify most of contradictions in UML class diagram using Z notation [1]. The Z notation was chosen for the various benefits that it offers. Hall [5][6] identifies several advantages of formal methods and concluded that they "contributes to demonstrably cost-effective development of software with very low defect rates". This study makes use of Anthony Hall's model [2][4] of specification and interpretation of class hierarchies to express UML [2] class diagram in Z [1]. The obtained model uncovers inconsistencies of a given UML class diagram. We selected Anthony Hall's model because it is referenced by most of works published to date and it models all needed concepts for the inconsistencies studied, namely: class hierarchy, multiplicity and association between classes. A prototype was devised to automatically generate formal specifications based on Anthony Hall's model [2][4]. All presented examples were automatically generated then type-checked with Z/EVES [13].

The structure of the rest of this paper is as follows. In Section 2, we present the related work. Section 3 provides a summary of Z notation [1] used in this paper. Section 4 summarizes Anthony Hall's model [2][4]. Section 5 illustrates how UML [2] inconsistencies identified in published previous studies are uncovered in Z [1]. Section 6 draws some conclusions and future works.

II. RELATED WORK

There are several researches that are closely related to our work. A method for the automatic detection of the contradictions in UML Class Diagram has been introduced in [10]. Two kinds of inconsistencies were detected: contradictory multiplicities and the disjoint constraint violation. A semantic of UML in terms of first order logic was used to translate the class diagram into a program in logic. Our work inspires by this approach and chooses to formalize all the UML class model into the Z notation, both contradictions studied in [10] trivially surfaced. The strength of our approach takes root into the simplicity and elegance of Anthony Hall's class hierarchies model. Also, the use of the Z notation made it possible to foster the Z/EVES [13] system for future investigations of the UML design robustness and to automatically process the model.

In [8], a definition of a production system language and rules specific to UML software designs is proposed. The system aims at detecting inconsistencies, notifying users and automatically fixing the inconsistency during the design process. The production system uses the Jess rule Engine [15]. In our approach, we use the Z notation [1] based upon set theory and mathematical logic. In the generated model, an inconsistency appears as two inconsistent predicates as it will be illustrated in Section 5. Our approach provides more visibility on the generated predicates, which enables further investigations on the software correctness. In the same context, the RoZ tool [11] has been developed to automatically generate formal specifications from UML class diagram. The UML design is completed by annotations in Z. However, this tool is different from ours, on the one hand, RoZ does not tackle inconsistency detection in UML class diagram. On the other hand, this tool requires the

 ISBN: 978-1-61208-165-6

designers to annotate in Z the UML class diagram to proceed with the generation of specifications, hence, the tool RoZ [11] requires Z specification at the UML stage, whereas in our method, we generate a complete translation of UML class diagram without any preliminary Z annotations. In our approach the separation of UML and Z allows to work on UML designs provided by software engineers without Z knowledge. When the model is generated, it offers the choice between an automatic processing to detect inconsistencies or a human static checking by a Z literate for further investigations.

In [16] and [17], a formal representation of UML models is proposed. The formal specification obtained is used to express and check some properties, called conjectures, on the model. Whereas in our approach we check the structural inconsistencies of a UML Class Diagram in general, focusing on generalization and multiplicities.

III. SUMMARY OF Z NOTATION

Z [1] 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. According to [12], a schema is a "structure describing some variables whose values are constrained in some way". A schema consists of two parts: the declaration part which contains the declaration of state variables and the 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. For example, the following schema denotes the type Rider, which is composed of four state variables with their types.

Rider
```
self: RIDER
name: NAME
weight: WEIGHT
skill: SKILL
```

At an early stage of the specifications, the new types are introduced as given sets. New introduced types serve as basic types in the specification. A given set is introduced between square brackets. For example, to introduce a given set named OBJECT, we write:

[OBJECT]

The symbol $\mathbb{P}$ is used to denote all subsets of a set. For example, to denote RIDER a subset of the set OBJECT we write:

$$RIDER: \mathbb{P}\, OBJECT$$

Several subsets can be defined at once, for example the following declaration

$$MAN, WOMAN: RIDER$$

introduces two subsets of *RIDER*. To denote that the two sets are disjoint we write

$$MAN \cap WOMAN = \varnothing$$

It could be abbreviated to:

$$disjoint \langle MAN, WOMAN \rangle$$

To denote a partial function named *idRider* from *RIDER* to *Rider* we write:

$$idRider: RIDER \rightarrow\!\!\!\!\!+ Rider$$

dom *idRider* denotes the domain of the partial function *idRider*, and ran *idRider* denotes its range. We can also define a function by set comprehension, example:

$$idRider = \{rider : Rider \bullet rider.self \mapsto rider\}$$

The function *idRider* is the set of all mappings $rider.self \mapsto rider$.

We can also define a function using lambda notation which is: (λ declaration | constraint $\bullet$ result)

For example, the relation f on the set of natural numbers $\mathbb{N}$ associates to each natural number m, the unique number $2*m+1$ as follow:

$$f: \mathbb{N} \leftrightarrow \mathbb{N}$$
$$f = (\lambda m:\mathbb{N} \bullet 2*m+1)$$

IV. UML CONSTRUCTS IN Z

In order to detect inconsistencies, a formal translation of UML constructs is used. The Z notation is the selected formal notation. The translation must meet a published model widely referenced. The model used is Anthony Hall's [2][4]. The most needed constructs are the class construct and the generalization relationship. The model is presented through the example of the riding school from [4] illustrated in Figure 1.

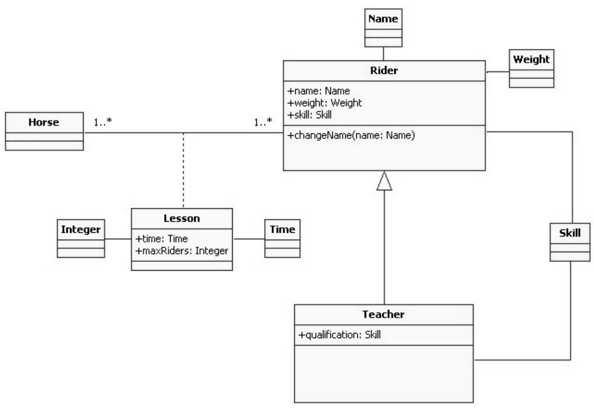

Figure 1. A Riding School UML Class Diagram

In Object-oriented modeling, a class describes the state and behavior of the class objects. The objects of a class are also called the class instances. The set of all object identities in Anthony Hall's model is introduced as the given set *[OBJECT]*. To model the set of all classes we introduce the given set *[CLASS]*. A class is a particular member of *CLASS*. In the example of the riding school, the UML rider class is translated into a schema containing the attributes and their types. An attribute *self* represents the identifier of the current instance. The name of the schema in Z is the concatenation of name of the class with the text '*CoreClass*'. Then a free type, with the same name as the class, is defined. It adds an optional nil value to be used in initializations.

In our example, a schema called *SRider* represents all instances of the class. The state variable *riders* represents the set of the riders identified by the system. The state variable *ridersIds* is the set of their identities. A function *idRider* binds each unique instance identifier to the corresponding rider.

$$RIDER : \mathbb{P}\ OBJECT$$

RiderCoreClass

$$\begin{aligned} &self : RIDER \\ &name : Name \\ &weight : Weight \\ &skill : Skill \end{aligned}$$

$$Rider ::= nilRider \mid ridercoreclasstorider\langle\langle RiderCoreClass\rangle\rangle$$

SRider

$$\begin{aligned} &riders : \mathbb{P}\ Rider \\ &idRider : RIDER \rightarrow\!\!\!\!\!\!\!\!\rightarrow\ Rider \\ &riderIds : \mathbb{P}\ RIDER \\ &\hline idRider = \{ ridercoreclass : RiderCoreClass \\ &\bullet\ ridercoreclass.self \mapsto ridercoreclasstorider\ ridercoreclass\ \} \\ &riderIds = \mathrm{dom}\ idRider \end{aligned}$$

An initialization schema is generated for each class to indicate the initial value of each attribute. Two types of initialization are proposed: an initialization by default which allows assigning nil values defined above to all attributes and the second method is used to initialize the attributes with values provided by the user.

InitRiderCoreClassByDefault

$$\begin{aligned} &RiderCoreClass' \\ &\hline name' = nilName \\ &weight' = nilWeight \\ &skill' = nilSkill \end{aligned}$$

InitRiderCoreClassWithValues

$$\begin{aligned} &RiderCoreClass' \\ &name? : Name \\ &weight? : Weight \\ &skill? : Skill \\ &\hline name' = name? \\ &weight' = weight? \\ &skill' = skill? \end{aligned}$$

The following example illustrates the way to formalize a method using the Z notation. Each method is translated into an operation schema. Each operation includes a schema that indicates whether the system state will be changed (RiderOp below) or remains unchanged (RiderGet below). This schema also guarantees us that the object identifier (self) remains unchanged.

Since the formal model is automatically generated from the UML Class Diagram, only the method signature is defined (ChangeNameRider below).

RiderOp

$$\begin{aligned} &\Delta RiderCoreClass \\ &\hline self' = self \end{aligned}$$

RiderGet

$$\begin{aligned} &\Xi RiderCoreClass \\ &\hline self' = self \end{aligned}$$

changeNameRider

$$\begin{aligned} &RiderOp \\ &name? : Name \end{aligned}$$

In Object-oriented programming, the setters/getters methods are often used. The setter method takes a new value as an input parameter to modify the private attribute. The getter method returns the value of the private attribute. In our tool, the getters/setters methods are automatically generated for each class.

setskillRider

$$\begin{aligned} &RiderOp \\ &skill? : Skill \end{aligned}$$

 ISBN: 978-1-61208-165-6

$$\mathrm{R}setskillRider: Skill \rightarrow Rider \rightarrow Rider$$

$$\begin{aligned}&\mathrm{R}setskillRider = \{\ skill: Skill \\ &\quad \bullet\ skill \mapsto \{\ setskillRider \mid skill? = skill \\ &\bullet\ (ridercoreclasstorider\ \theta\ RiderCoreClass \\ &\mapsto ridercoreclasstorider\ \theta\ RiderCoreClass')\ \}\ \}\end{aligned}$$

setskillRiderSystem

$$\begin{aligned}&\Delta SRider \\ &rider?: RIDER \\ &skill?: Skill \\ &\hline idRider' = idRider \oplus (\{rider?\} \lhd idRider \fatsemi \\ &\mathrm{R}setskillRider\ skill?)\end{aligned}$$

getskillRider

$$\begin{aligned}&RiderGet \\ &skill!: Skill \\ &\hline skill! = skill\end{aligned}$$

The example below illustrates the transformation rule of the inheritance relationship between Teacher Class that inherits from Rider Class. The inheritance relationship between two classes is translated into Z by the inclusion of the schema of the super-class in the declaration part of the schema of subclass. In any inheritance relationship, the set of object identities of the subclass is a subset of the object identities of the super-class. To express this relationship, we define the schema called RiderTeacherHierarchy. The lambda function (λ Teacher • θ Rider) used in the predicate part of RiderTeacherHierarchy denotes the projection function from Teacher state to Rider state.

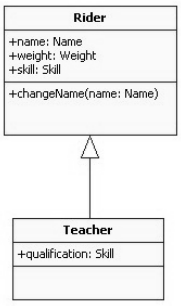

Figure 2. Example of Inheritance Relationship

TeacherCoreClass

$$\begin{aligned}&RiderCoreClass \\ &qualification: Skill \\ &\hline self \in TEACHER\end{aligned}$$

$$\begin{aligned}&Teacher ::= nilTeacher \\ &\mid teachercoreclasstoteacher\ \langle\!\langle TeacherCoreClass \rangle\!\rangle\end{aligned}$$

STeacher

$$\begin{aligned}&teachers: \mathbb{P}\ Teacher \\ &idTeacher: TEACHER \rightarrow Teacher \\ &teacherIds: \mathbb{P}\ TEACHER \\ &\hline idTeacher \\ &\quad = \{\ teachercoreclass: TeacherCoreClass \\ &\qquad \bullet\ teachercoreclass.self \\ &\qquad \mapsto teachercoreclasstoteacher\ teachercoreclass\ \} \\ &teacherIds = \mathrm{dom}\ idTeacher\end{aligned}$$

RiderTeacherHierarchy

$$\begin{aligned}&SRider \\ &STeacher \\ &\hline teacherIds = riderIds \cap TEACHER \\ &\forall\ t: teacherIds \\ &\bullet\ (\lambda\ TeacherCoreClass \bullet\ \theta\ RiderCoreClass) \\ &(teachercoreclasstoteacher\ \tilde{}\ (idTeacher\ t)) \\ &\quad = ridercoreclasstorider\ \tilde{}\ (idRider\ t)\end{aligned}$$

To get an overview of all classes of the system and relationships that bring them together, the schema System is introduced.

System

$$\begin{aligned}&\mathbb{S}Rider \\ &\mathbb{S}teacher \\ &RiderTeacherHierarchy\end{aligned}$$

This model is used in Section 5 to illustrate how common UML inconsistencies [9][10] are translated into inconsistent predicates.

V. UML INCONSISTENCIES IN Z

In this section, a formalization of UML inconsistencies is presented using Z Notation. This formalization is based on the model presented in Section 4. Each inconsistency is presented through an illustrative example previously published.

A. Generalization and Disjointness

In a UML Class Diagram, the disjoint constraint means that an instance of the super-type may not be a member of more than one sub-type, it is denoted in UML between brackets near the inheritance arrow, i.e., multiple inheritance of disjoint classes is forbidden. The example studied in the papers [9][10] and illustrated in Figure 3. shows a diagram where {disjoint} constraint is violated.

 ISBN: 978-1-61208-165-6

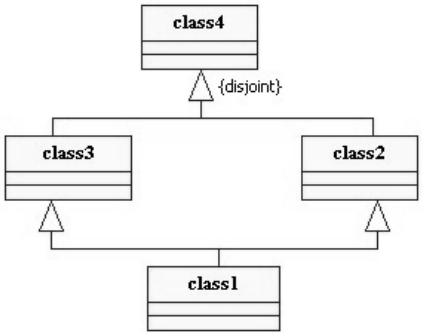

Figure 3. Inconsistent Class Diagram

The formalization of disjointness is given by the inclusion of a predicate which guarantees the disjointness of classes mentioned with the constraint {disjoint}.

System

$Sclass1$
$Sclass3$
$Sclass2$
$Sclass4$
class3class1Hierarchy
class2class1Hierarchy
class4class2Hierarchy
class4class3Hierarchy

---

disjoint ⟨ *class2Ids* , *class3Ids*⟩

The predicate *disjoint*⟨*class2Ids* , *class3Ids*⟩ is equivalent to the predicate:

$$class2Ids \cap class3Ids = \varnothing \quad (1)$$

The constraints

$$(class1Ids = class2Ids \cap CLASS1) \wedge (class1Ids = class3Ids \cap CLASS1) \quad (2)$$

are introduced in the schema *System* from *class2class1Hierarchy* and *class3class1Hierarchy*.

(1) And (2) implies that $class1Ids = \varnothing$ (3)

If *class1* is instantiated then $class1Ids \neq \varnothing$, hence the inconsistency.

To check the consistency of the specification, a disjointness theorem is generated when a disjoint property is reported on the UML model.

theorem *disjointness*
$\exists\, Sclass1 \mid class1Ids \neq \varnothing \bullet System$

If the theorem cannot be proved, then the System is inconsistent.

B. Completeness and Disjointness

We found also in the UML class diagram the {complete} constraint.

The {complete} constraint means that each instance of the super-type must be a member of one of the sub-types. Here is an example from [9].

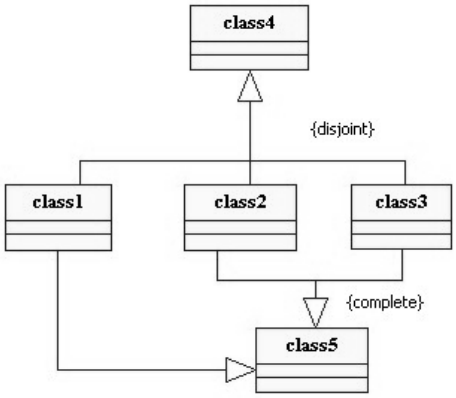

Figure 4. Inconsistent Class Diagram

The {complete} constraint used in the Figure 4. imposes on the *class5* to be specialized either as *class2* or *class3*. The completeness constraint is translated by the following predicate in the system schema:

$$class5Ids = class2Ids \cup class3\text{Ids}$$

This predicate expresses that all instances of *class5* belong either to *class2* or *class3*. The system obtained is:

System

$Sclass1$
$Sclass2$
$Sclass3$
$Sclass4$
$Sclass5$
class4class1Hierarchy
class4class2Hierarchy
class4class3Hierarchy
class5class1Hierarchy
class5class2Hierarchy
class5class3Hierarchy

---

disjoint ⟨*class1Ids*, *class2Ids*, *class3Ids*⟩
$class5Ids = class2Ids \cup class3\text{Ids}$

On the one hand in the final schema System, the predicate

disjoint ⟨*class1Ids*, *class2Ids*, *class3Ids*⟩

translates the disjoint constraint in UML and the predicate

$class5Ids = class2Ids \cup class3\text{Ids}$

translates the complete constraint in UML.

 ISBN: 978-1-61208-165-6

On the other hand, the generalization between *class1* and *class5* is translated into the following predicates:
$class1Ids = class5Ids \cap CLASS1$ from the schema *class5class1Hierarchy*.
where $class5Ids: \mathbb{P} CLASS5$ from the schema $\mathcal{S}class5$.
Therefore $class1Ids \subseteq CLASS5 \cap CLASS1$ implies that $class1Ids = \emptyset$.
If *class1* is instantiated, then $class1Ids \neq \emptyset$. Hence the inconsistency.
The following theorem is used to check the consistency of the schema *System* when using disjoint and complete constraints simultaneously.

theorem *completeness*
$\exists Sclass1 \mid class1Ids \neq \emptyset \bullet System$

In the same way, it detects when there is no such a *System*.

C. Multiplicities

Consider the following class diagram used in the article [9][10] representing a multiple inheritance:

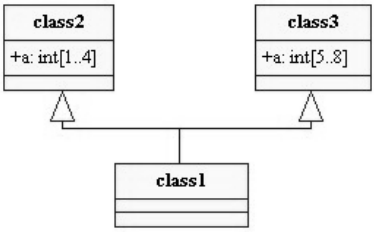

Figure 5. Inconsistent Class Diagram

In this example, we have three classes named *class1*, *class2* and *class3*. *Class1* inherits from both *class2* and *class3*. The multiplicity of an attribute indicates the number of values that attribute can contain. In the example, *class2* has an attribute with a multiplicity maximum of 4 and minimum of 1, *class3* has an attribute with the same name but different bounds: the multiplicity maximum is 8 and minimum is 5.

The following schema illustrates the multiplicity formalization:

class2
$self: CLASS2$
$a: \mathbb{P}\mathbb{Z}$

$1 \leqslant \# a \leqslant 4$

class3
$self: CLASS3$
$a: \mathbb{P}\mathbb{Z}$

$5 \leqslant \# a \leqslant 8$

We use Anthony Hall's modelling [2][4] summarized in Section 4 to represent the inheritance relationship. We include the schema of the super-class in the declaration part of the schema of subclass. In this example, we have a multiple inheritance. *class1* is represented by the following schema:

class1
class3
class2

$self \in CLASS1$

In Z, the introduction of a schema S1 into another schema S2 introduces all the state variables and predicates of S1 into S2. In this example, the inconsistency is immediately detected in Z because class1 inherit two attributes with the same name from two different super-classes class2 and class3. The Z/EVES [13] immediately uncovers such a redundant declaration. Even if we keep only one declaration in the variable part of the schema, the two predicates remain inconsistent. It is worth saying here that the UML standard [2] is ambiguous in the case of multiple inheritance of the same attribute. Therefore, it is up to the designer to provide a semantic in that case.

VI. Conclusion and Future Work

A. Concluding Remarks

This article illustrated most frequent UML inconsistencies published so far using Anthony Hall's model [2][4] most of these are translated into contradictory predicates. In some ambiguous cases, UML must be supplemented by an additional formal semantic. Typically UML lacks a semantic for multiple inheritance of attributes with the same name. A prototype has been developed to automatically translate UML designs into their formal counterpart. The Z notation makes the formal translation of the design particularly suitable for further investigation in Z.

B. Future Work

There are two ways to build on this work. First we are developing an automated and interactive verifier of the inconsistencies using Z/EVES [13] meanwhile a formalization of the Object Constraint Language is prepared in order to translate UML Class Diagrams using OCL [14] into a more precise Z counterpart. The current prototype is completed to automatically generate formal specifications from UML Class diagrams annotated by OCL constraints.

References

[1] J. M. Spivey: The Z Notation: A Reference Manual, Prentice Hall, Englewood Cliffs, NJ, Second Edition, 1992.

[2] Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.3, May 2010. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/. Sept 11, 2011.

 ISBN: 978-1-61208-165-6

[3] A. Hall, "Using Z as a Specification Calculus for Object-Oriented Systems". In Bjorner D, Langmaack H (eds), Proceedings of VDM 90, Lecture Notes in Computer Science No. 428, pp. 290 - 318. Springer Verlag, 1990.

[4] A. Hall, "Specifying and Interpreting Class Hierarchies in Z", Z User Workshop, Cambridge 1994, ed. J. P. Bowen and J. A. Hall, Springer, 1994.

[5] A. Hall, "Realising the Benefits of Formal Methods", Formal Methods and Software Engineering, LNCS 3785, Springer, pp. 1-4, 2005.

[6] A. Hall, "Seven Myths of Formal Methods", IEEE Software, September 1990, pp. 11-19.

[7] B. Nuseibeh, S. Easterbrook and A. Russo, "Leveraging Inconsistency in Software Development". IEEE Computer, vol. 33, pp. 24–29, April, 2000.

[8] W. Liu, S. Easterbrook and J. Mylopoulos: Rule-based Detection of Inconsistency in UML Models; Proc. Workshop on Consistency Problems in UML-Based Software Development, pp. 106-123, 2002.

[9] K. Kaneiwa and S. Satoh, "Consistency Checking Algorithms for Restricted UML Class Diagrams". In Proceedings of the Fourth International Symposium on Foundations of Information and Knowledge Systems, vol. 3861, pp.219-239, 2006.

[10] K. Satoh, K. Kaneiwa and T. Uno, "Contradiction Finding and Minimal Recovery for UML Class Diagrams using Logic Programming". Proceeding of 21st IEEE International Conference on Automated Software Engineering (ASE'2006), pp. 277-280, 2006.

[11] Yves Ledru. RoZ tool. 22 february 2000. http://vasco.imag.fr/RoZ/index.html. Sept 11, 2011.

[12] Jim Woodcock and Jim Davies.: Using Z Specification, Refinement, and Proof. University of Oxford, 1995.

[13] Irwin Meisels. Software Manual for Windows Z/EVES Version 2.3. ORA Canada Technical Report TR-97-5505-04h, June 2004.

[14] Object Management Group (OMG). Object Constraint Language. Version 2.2, February 2010. http://www.omg.org/spec/OCL/2.2/PDF/. Sept 11, 2011.

[15] Jess, the Rule Engine for the JavaTM Platform. http://www.jessrules.com/. Sept 11, 2011.

[16] N. Amalio, F. Polack and S. Stepney. "UML + Z: UML augmented with Z". In Software Specification Methods: an Overview Using a Case Study. Marc Frappier and Henri Habrias, editor. Hermes Science Publishing. 2006.

[17] N. Amalio, S. Stepney and F. Polack, "Formal Proof from UML Models". In et al, J. D.,ed., ICFEM 2004, volume 3308 of LNCS, pp. 418–433. Springer .2004.

 ISBN: 978-1-61208-165-6