Formalizing UML Software Models of Safety Critical Systems

Sophine Clachar and Emanuel S. Grant
Department of Computer Science
University of North Dakota
Grand Forks, ND 58202
sophine.clachar@und.edu

Abstract

The Unified Modeling Language (UML) is the de facto standard for designing models of software systems in both industry and academia. UML has many advantages, and is often the tool of choice when conveying information between various stakeholders. UML's main disadvantage is that it is too abstract – resulting in ambiguous models. In safety critical systems, ambiguity could result in the loss of property or be detrimental to life. With the continuous use of UML in the software industry, there is a need to amend the informality of software models produced.

The objective of this research is to use formal specification to enhance the shortcomings of UML and analyze its significance to safety critical systems. The proposed approach is to design a UML class diagram of a safety critical system and remodel it using formal methods. From this process, an assessment can be made of the inherent benefits of formalizing models of safety critical systems.

1. INTRODUCTION

Graphical models of software systems are designed in the early phase of the Software Development Life Cycle (SDLC). In the software industry, models are important because they: 1) serve as a blueprint of the proposed system, 2) aid in the understanding of the project, and; 3) act as a guideline for developers. The nature of the proposed system will determine how complex its models will be and the tactics used to design and test them. Safety critical systems can be regarded as a complex system and models of complex systems are built because one cannot comprehend any such system in its entirety [1]. Models serve as an abstract view of the system and suppress any details pertaining to implementation. Its purpose is to represent the system at a high-level; and in achieving this level of abstraction, it is possible to over-look or misrepresent critical aspects of the system. Therefore, it is important that the correctness of software models be ascertained at an early stage in the software development life cycle – especially if these models are of safety critical systems.

A safety critical system is one in which any loss or misrepresentation of data could result in injury, loss of lives and/or property. This type of system is common in industries such as:

- Aeronautics:- systems used to regulate the flight of aerial vehicles,
- Medicine:- systems which diagnose and treat patients,
- Space Exploration: - systems that transport or support life form and objects into outer-space.

The software development community has recognized the Unified Modeling Language (UML) [1] as the de facto standard for designing graphical models of software systems in both industry and academia. UML has many strong points. It is technical enough to model a system's internal and external components; yet simple enough to explain to non-technical stakeholders – such as customers. UML has a wide variety of models and notations that is intended to equip the developer with the appropriate tools to capture the static and dynamic aspects of a system. Its flexibility and object oriented modeling capability are among the primary reasons for its acceptance. However, its weaknesses have posed many challenges and ambiguity among software developers.

UML's disadvantage lies in the lack of rigorous rules and precise semantics when designing models [3]. For this reason, many decisions on how user requirements are modeled are left to the modeling technique adopted by the software engineer. Ambiguity is caused when these models are circulated among software engineers and each interprets them differently. To overcome prominent limitations of UML, formal specification techniques have been proposed by [4] and are often used to describe and verify models. Formal methods involve the use of a specification language to design models that are mathematically tractable and unambiguous. To amend the informality of graphical models, model transformation will be done and the formal models will be analyzed by a proof tool to check its syntax and semantics. Any errors found will be amended to the original graphical models; and this process will be repeated until the proof tool does not detect any errors in the model. The specification language that will be used is called Z

(“Zed”) [2]. Z is a notation that is used in describing software systems based on the mathematical principles of set theory and predicate logic. It was created by Jean-Raymond Abrial [3] in 1977.

This paper encompasses the development of a UML class diagram of a safety critical system. This aspect of the work was inspired from research currently being conducted at the University of North Dakota (UND). The focus of their research is on designing an air-truth system that acts as a guide for the operation of unmanned aerial vehicles (UAVs) in the US National Airspace [9]. In such systems, the integrity and correctness of data is crucial to its operation and acceptance by, not just the Federal Aviation Administration (FAA), but by all interested parties. In the realm of software development, no perfect software development strategy exists. However, finding an optimal approach to a particular application domain is fundamental to acceptance. In the design of safety critical system, its very nature requires that an optimal methodology and technique be sought and applied – especially if a loss in life or property may occur.

The remainder of the report is as follows; Section 2 presents the background research areas, Section 3 outlines the proposed methodology and Section 4 concludes the report.

2. Background

2.1 The Unified Modeling Language

The UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems [1] and serves as a blueprint for software engineers. UML is an object-oriented modeling language that promotes some of the best software development practices; and this very quality is among the primary reasons for its acceptance. UML's usefulness is felt in the early phase of the software development life cycle (SDLC) where it is used to depict a high-level representation of the proposed system.

The UML helps developers to obtain an abstract view of the proposed system. This abstract representation is achieved through the design of various types of models, which capture the structure and behavior of the system and its internal and external components. Its intent is to facilitate improved communication among members of the development team as each can comprehend the project as a whole before dividing the work, and; it helps developers to identify if user requirements will be adequately addressed by the system. UML is widely accepted because of its simplicity, which makes it easily understood by developers thereby making it easily conveyed to their customers [6].

2.1.1 UML Diagrams and Relationships

Diagrams in UML are grouped into the following three categories: structure, behavior, and interaction diagrams. Structure diagrams represent the static composition of the system [5]. These diagrams illustrate the role of objects and classes – along with their respective attributes and operations. They also show the flow of data between objects and classes; and the relationships that exist between them. Examples of structure diagrams include class, component, object, deployment and package diagrams.
Behavior diagrams represent the dynamic features of the system by showing how the system is acted upon. Behavior diagrams include use case, activity, and state diagrams. Interaction diagrams are an extension of behavior diagrams but focuses mainly on the internal elements of the system. Examples of interaction diagrams include sequence and collaboration diagrams.

Class diagrams and use case diagrams facilitate communication between non-technical stakeholders (i.e. customers) and developers. The more complex UML diagrams such as sequence and state chart diagrams are more technical and suitable for astute stakeholders; such as engineers and developers.

The scope of this paper will be on the static UML models – more specifically, the class diagram. Creation of a new class diagram in UML begins with a class. By UML standards, a class is represented as a rectangular box with three compartments: the class header, list of attributes, and list of operations. For demonstration purposes, classes will be depicted with two sections: a header and a suppressed list of attributes and operations. Figure 1 illustrates a UML class diagram, in which the generalization/specialization relationship is represented, with classes *Class B* and *Class C* are specialized classes of the generalized class *Class A*.

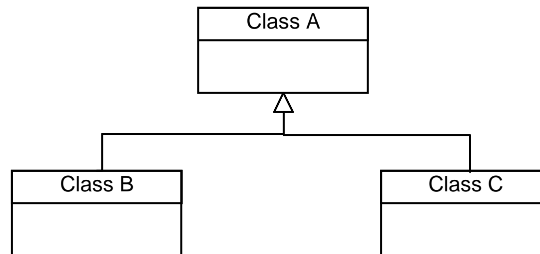

Figure 1: Example of a UML class diagram

In the UML, class diagrams relationships are depicted by lines that connect two or more classes. These lines specify the types of relationships that exist between the classes, the flow of information, and restrictions on the relationships. Relationships include, but are not limited to, associations (bidirectional and unidirectional), aggregations (strong and weak) and hierarchical (generalization/specialization, or parent-child). Figure 2 illustrates a class diagram with a simple association between classes *Class A* and *Class D*,

and an aggregation relationship between classes *Class A*, *Class B*, and *Class C* – Wherein class *Class A* is composed of classes *Class B* and *Class C*.

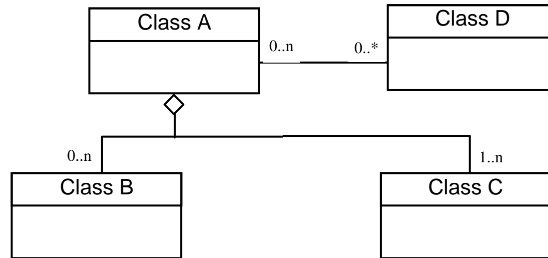

Figure 2: Example of UML class diagram relationship

2.1.2 Disadvantages of UML

UML, like many other software development tools, has its weaknesses. UML lacks precise semantics, which results in the models being subject to multiple interpretations. Due to UML's innate flexibility, developers are given much leeway when designing models. This is both positive and negative. This freedom enables the developer to capture and model requirements based on the modeling technique they have adopted. Problems arise when these models are circulated among the development team and each developer interprets the models incorrectly – which could affect the latter stages of the SDLC. Annotations can be used to alleviate this; however, annotations can be misinterpreted because it is expressed in natural language [3].

Another disadvantage is, after the initial stage of the project, updating models is often deemed tedious and time-consuming. As a result, critical changes are often not reflected in the model; albeit the source-code reflects the change. Therefore, when the software maintenance is required and developers need to obtain a general overview of the project, the UML models are often inconsistent with the source code and its significance is lost. [7]

In some systems, these disadvantages may not have a significant effect on the quality of software produced. However, in safety critical systems a flaw could result in the loss of property or be detrimental to life. Since UML is widely accepted within the industry, there is a need for methods to test the correctness of its models. This can be achieved with the use of formal specification techniques (FSTs).

2.2 Formal Specification Techniques

Formal specification involves the use of a specification language to describe and model software more accurately. It uses mathematical concepts and principles to model both the static and dynamic aspects of a system; which results in software models that are not just sound but tractable. FSTs allow developers to analyze the syntax and semantics of models using a proof tool and make any necessary changes in an evolutionary manner. The specification language that will be used is called Z. To transform UML models into Z notation, a Z schema is created for each UML class, association and relationship. The attributes and operations of the UML class are also included in the schema. Constraints are defined on the relationships between schemas. This prohibits or permits a schema access to its environs.

A schema in Z has two parts: a declaration and a predicate part [11]. The declaration part consists of variables which are synonymous to the definition of attributes in a UML class. The fundamental difference, however, is that the variable declaration types are expressed as mathematical notations unless they are user defined types. The predicate part imposes constraints on the variables defined in the declaration part. Figure 3 illustrates the structure of a Z schema.

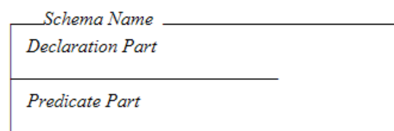

Figure 3: Example of a Z schema

After the models are transformed into Z notation, they will be analyzed by Z/EVES. Z/EVES is a proof tool which tests the syntax and semantics of the models. This process is called software testing. Software testing is the process by which software models undergo a series of analysis to check for errors. It is also used to determine if the quality of the software produced meets user requirements and if it performs as expected – even under stress. It is impractical for testing to detect all types of errors; and even the most rigorous testing procedure will, as Edsger Dijkstra stated, show the presence of bugs but never their absence [8]. However, it is still important for software models to undergo testing – especially if they are models of a safety critical system.

2.3 Model Transformation

The level of abstraction provided by models helps developers and stakeholders visualize different aspects of the system while avoiding the details of implementation. For any given system, a large number of models can exist and it is important to ensure their overall consistency [10]. Model transformation uses a set of rules called transformation

rules, which accepts one or more models as input and produce one or more target models as output [10].

Model transformation can be either manual – i.e. defining custom transformation rules; or it can be automatic – applying predefined transformation rules. It is important, however, that the software engineer have a good understanding of the scope of the project, the syntax and semantics of the source and target models irrespective of the transformation approach taken. In order to automate the proposed approach, transformation rules will be designed and applied to the models. The source models will be the UML class diagrams and the target models will be their equivalent Z schemas.

3. Methodology

There is a plethora of literature in the area of transforming UML class diagrams using FSTs. There are many similarities as well as differences in each approach. However, this research seeks to modify an existing approach and apply it to a real world system. This research will also use a theorem prover to verify the formal models in order to determine if the approach is sound. From this analysis, one can determine the best way to automate the formalization of safety critical systems. The proposed approach is inspired collectively by the works of [3],[4] and [11]. The highlight of their work is on the advantages and disadvantages of a more formal approach to modeling software systems. It also demonstrates how to transform UML class diagrams into formal models using Z.

For the purpose of automating this process, the approach taken in this methodology is to define a strict set of sequential rules that – if followed correctly – will achieve correct formal models. The fundamental difference between the two approaches is that the modified approach will be applied as a case study against a real world safety critical system, inspired by research at UND. From this case study, one can determine the suitability of automation and the optimality of the approach taken.

Figure 4 highlights the proposed approach. Initially, UML class diagrams will be created for the system. Model transformation will then take place producing representative formal models. The formal models will be analyzed using Z/EVES to check the syntax and semantics of the models. If errors are found, they will be documented and corrections will be made to the original graphical models. The corrected models will undergo model transformation, and the transformed models will be analyzed by the proof tool; this process will be repeated until errors are not detected by Z/EVES. Subsequently, programming and code generation can begin.

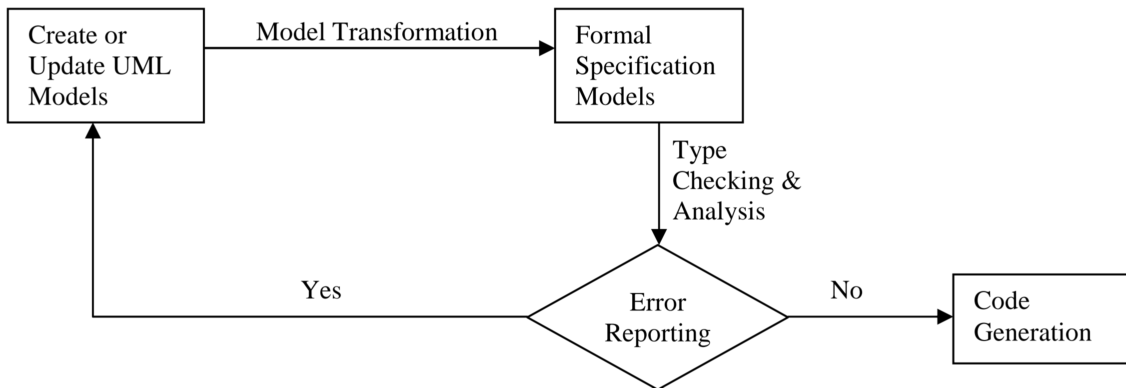

Figure 4: The Transformation Process

An example of the UML class diagram which was derived from the real world system is illustrated in Figure 5. Figure 5 illustrates the *Aircraft* class as being composed of a *Coordinate* class and specialized as a *MAV* (Manned Arial Vehicle) and *UAV* (Un-manned Arial Vehicle). These models will be transformed into Z notation then analyzed by use of the Z/EVES tool; any corrections found will be amended to the original UML models.

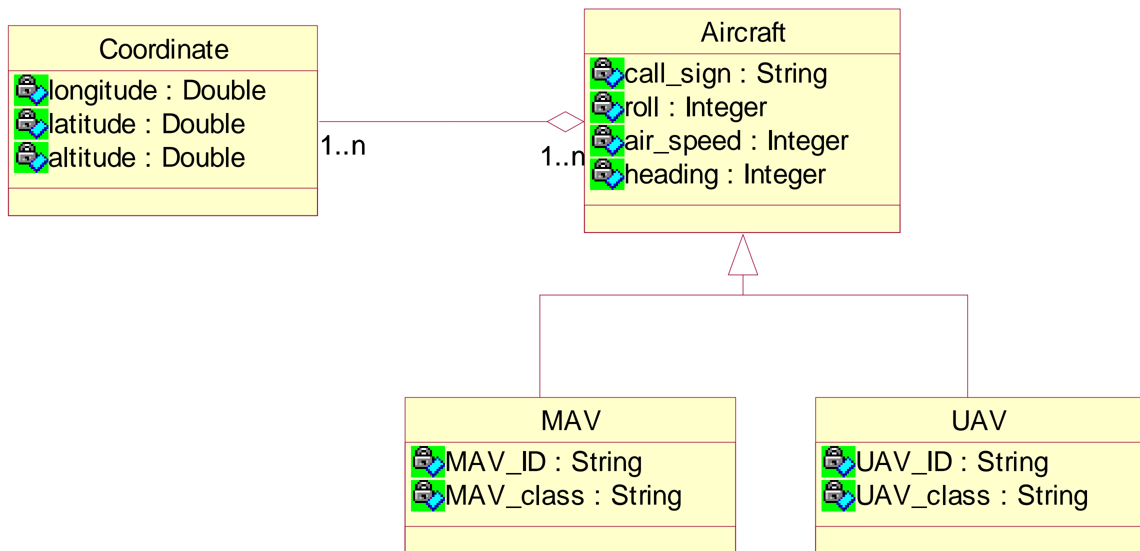

Figure 5: UAS Aircraft Class Diagram

The set of rules for the model transformation are as follows:

A schema will be created for each of the following:

- Attributes:- The attributes of a class will be grouped together. Predicates will be defined for constraints on each attribute. In Figure 5, the *logitude*, *latitude* and *altitude*

attributes will be grouped together in a schema. This process will be performed for each class.

- Classes:- A schema will be created for each UML class which will be comprised of their attribute schema and operations. In Figure 5, a schema will be created for the *Coordinate* class – which will include its attribute schema. This process will be applied to all UML classes.

- Relationships:- This schema definition pertains to the different types of relationships that exists in the class diagram, along with their respective multiplicities. A schema will be created for each relationship. In Figure 5, two relationship schemas will be defined. One for the aggregate relationship and the other for the generalization/specialization relationship. Each schema will define additional constraints on the relationship, such as the multiplicities and schema ownership rules. The relationship schema will also define constraints such as the number of objects that are allowed to be instantiated in the system at any given time.

One of the key features of this approach is the processing of UML annotations. To avoid ambiguities, software engineers often attach annotations to their graphical models. However, anything expressed in natural language is innately subject to multiple interpretations. Therefore, this work also seeks to formalize constraints and annotations imposed on UML graphical models.

After the formal models are created, they will be analyzed by use of the Z/EVES theorem prover. This process is essential for identifying errors and omissions in the original graphical models. If errors are found, they will be documented for the software engineer to make changes to the UML models. This process will be repeated until errors are not detected in the models. An example of the formal models produced by the manual transformation of the graphical model in Figure 5 is illustrated below:

```
[RANGE_OBJECT]

[COORDINATE]

[STRING]
```

```
── Aircraft ──────────────────────────
aircraft: ℙ RANGE_OBJECT
call_sign: RANGE_OBJECT → STRING
air_speed: RANGE_OBJECT → ℕ
heading: RANGE_OBJECT ⤔ ℕ
──────────────────────
dom call_sign = aircraft
dom air_speed = aircraft
dom heading = aircraft
∀ airspeed: ℕ • airspeed ∈ ran air_speed ⇒ airspeed ⩽ 250
∀ hdg: ℕ • hdg ∈ ran heading ⇒ hdg ⩽ 360
──────────────────────────────────────
```

```
── Coordinate ────────────────────────
coordinate: ℙ COORDINATE
longitude: ℙ ℤ
latitude: ℙ ℤ
altitude: ℙ ℤ
air_coord: (ℤ × ℤ) × ℤ ⇸ COORDINATE
──────────────────────
dom air_coord = (longitude × latitude) × altitude
∀ alt: altitude • alt ⩽ 18000
∀ lat: latitude • - 90 ⩾ lat ⩽ 90
∀ lon: longitude • - 180 ⩾ lon ⩽ 180
──────────────────────────────────────
```

```
── MAV ───────────────────────────────
Aircraft
mav: ℙ RANGE_OBJECT
mav_ID: 𝔽 STRING
mav_class: 𝔽 STRING
aircraft_ID: RANGE_OBJECT → STRING
──────────────────────
dom aircraft_ID = aircraft
ran aircraft_ID = mav_ID
──────────────────────────────────────
```

```
── UAV ───────────────────────────────
Aircraft
uav: ℙ RANGE_OBJECT
uav_ID: 𝔽 STRING
uav_class: 𝔽 STRING
aircraft_ID: RANGE_OBJECT → STRING
──────────────────────
dom aircraft_ID = aircraft
ran aircraft_ID = uav_ID
──────────────────────────────────────
```

```
── Aircraft_Hierarchy ────────────────
MAV
UAV
uav1, mav1: ℙ RANGE_OBJECT
──────────────────────
∀ i, j: MAV • i.mav_ID = j.mav_ID ⇔ i = j
∀ i, j: UAV • i.uav_ID = j.uav_ID ⇔ i = j
∀ u: uav1; m: mav1 • u ∈ aircraft ∧ m ∈ aircraft ∧ u ∈ uav ∧
m ∈ mav
mav ∩ uav = ∅
mav ∪ uav ⊂ aircraft
──────────────────────────────────────
```

```
── Aircraft_Coordinate_Rel ───────────
Coordinate
Aircraft
rel: COORDINATE ↔ RANGE_OBJECT
──────────────────────
∀ i, j: Coordinate • i.coordinate = j.coordinate ⇔ i = j
dom rel = coordinate
ran rel = aircraft
∀ a: aircraft • 0 ⩽ # (rel ˜ ( {a} )) ⩽ 1
∀ c: coordinate • # (rel ( {c} )) ⩾ 1
──────────────────────────────────────
```

Figure 6: UAS Aircraft Z Schemas

The collection of schemas in Figure 6 depicts the formal model of the class diagram represented in Figure 5. From the aircraft class, an *Aircraft* schema was derived. In the first half of the *Aircraft* schema, variables were declared, however primitive data types were not used. In the second half of the *Aircraft* schema, constraints on the variables were defined; a similar approach was taken for the coordinate class in Figure 5.

The children of the aircraft class, i.e. MAV and UAV, where defined in terms of its parent *Aircraft* schema. This relationship is shown, where each child includes the parent schema in the variable declaration part, and in the predicate part constraints are defined

on the relationship. The *Aircraft_Hierarchy* schema further constrains this relationship by describing the uniqueness of the children and their objects. A similar approach was taken for the schema definition of the *Coordinate* class and its relationship with the *Aircraft* class.

The syntax and semantics of the schemas in Figure 6 were checked by use of the Z/EVES analysis tool. The proof tool essentially determined the validity of the schemas and their respective constraints. Any errors found were amended to both the original graphical models and the formal models.

The schemas in Figure 6 are cumbersome because the process applied in defining them did not adhere to the sequence of steps outlined earlier. Therefore, the automated transformation will not only speed up the formalization process but also simplify the schema definitions.

4. Conclusion

In the software development industry, the benefits of formal methods are known. However, unlike its counterpart graphical models, it is not quickly gravitated to because it is not very easy to learn. In order to circulate formal models among the software development team, each member is required to be adept in not just the area of formal methods but the specification language chosen. For this reason, the use of FSTs is not entertained unless it is deemed absolutely necessary. One such case is in the development of safety and mission critical systems.

Currently, formalization is conducted manually. In order to move from research to productive use of this technique, there has to be some high degree of automation. Therefore, conducting a case study in the area of automated tools for FSTs in safety critical systems will enlighten researchers on the complexity, advantages and possible use of such software.

In conclusion, the use of FSTs can be advantageous in the development of complex software systems. FSTs have existed before the conception of UML, its graphical counterpart. However, unlike UML it does not have that high level of simplicity that makes its models easily conveyed to both technical and non-technical stakeholders. It also requires a certain level of detail in order to exploit its full potential. Therefore, this case study will determine an optimal approach in the design of tools, which automate FSTs, their advantages and possible use.

References

[1] OMG, UML Language Specification, version 2.01, 2005

[2] Spivey, J. M. 1989 The Z Notation: a Reference Manual. Prentice-Hall, Inc.

[3] M. Shroff, R.B. France, "Towards a formalization of UML class structures in Z," compsac, pp.646, COMPSAC '97 - 21st International Computer Software and Applications Conference.

[4] R. France, A. Evans, K. Lano, B. Rumpe, The UML as a formal modeling notation, Computer Standards & Interfaces, Volume 19, Issue 7, 2 November 1998, Pages 325-334, ISSN 0920-5489, DOI: 10.1016/S0920-5489(98)00020-8.

[5] Bell, D. UML Basics: The Class Diagram. (09/15/2004). Retrieved from http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

[6] Hai, H., Yi-fang, Z., and Chi-lan, C. 2005. Unified Modeling of Complex Real-Time Control Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1* (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 498-499. DOI= http://dx.doi.org/10.1109/DATE.2005.322.

[7] K. Berkenkotter. Using uml 2.0 in real-time development: A critical review. *Proceedings of SVERTS workshop*, 2003.

[8] O. J. Dahl, E. W. Dijkstra, and C. A. Hoare, Eds. 1972 *Structured Programming*. Academic Press Ltd

[9] Stansbury, R. S., Vyas, M. A., and Wilson, T. A. 2009. A Survey of UAS Technologies for Command, Control, and Communication (C3). *J. Intell. Robotics Syst.* 54, 1-3 (Mar. 2009), 61-78. DOI= http://dx.doi.org/10.1007/s10846-008-9261-2.

[10] Sendall, S.; Kozaczynski, W., "Model transformation: the heart and soul of model-driven software development," *Software, IEEE* , vol.20, no.5, pp. 42-45, Sept.-Oct. 2003
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1231150&isnumber=27576

[11] France, R. 1999. A problem-oriented analysis of basic UML static requirements modeling concepts. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, United States, November 01 - 05, 1999). A. M. Berman, Ed. OOPSLA '99. ACM, New York, NY, 57-69. DOI= http://doi.acm.org/10.1145/320384.320390