A New Approach to Software Development Process with Formal Modeling of Behavior Based on Visualization

Abbas Rasoolzadegan, Ahmad Abdollahzadeh Barfourosh
Information Technology and Computer Engineering Faculty
Amirkabir University of Technology (Tehran Polytechnic)
{rasoolzadegan, ahmad}@aut.ac.ir

***Abstract*—This work investigates the advantages and limitations of various modeling methods. Despite of their advantages, due to some limitations of each modeling method, using only one of them as the sole approach will not ensure high quality software. This work proposes a new feasible approach to improve the software development process by integrating semi-formal and formal modeling methods. In this approach, software is initially modeled using the formal specification language Object-Z. The formal models, produced by Object-Z, are formally refined to ensure correctness. Then, software behavior is extracted and visualized in specific intervals using UML. Applying design patterns to the visualized models increases reusability and flexibility. The newly improved models are then re-formalized. Such an iterative and evolutionary process continues until developing the software with the desired quality. This paper proposes a new approach to develop reliable, yet flexible software.**

Keywords-Formalization; visualization; design patterns; formal modeling methods; semi-formal modeling methods.

I. INTRODUCTION

Requirements engineering (RE) plays a crucial role in software development cycle. Studies show that the major causes of most software projects failure are imprecise and incomprehensive understanding, elicitation, specification, analysis, validation, and verification of software requirements during software development process [3]. Moreover, mainstream software development, with its recurring practice of trial and error, already suffers from its premature insistence on code and program testing. The problem is that code is expensive; it has too much detail, and is not at the right level of abstraction to help thinking about the problem and design of its solution [1].

The increasing importance of requirements engineering and need for further abstraction leads to increasing use of models during software development cycle, in general, and throughout RE process, in special. Models can be used at different phases of a software life-cycle, ranging from requirements (more abstract) to detailed design (more concrete). It also gives a basis for a stepwise approach to software development: abstract models are refined into more concrete ones in a stepwise manner, where each step carries some design decisions. This is known as model refinement [3].

Models and modeling play a crucial role in software development cycle. In software engineering, models are used to describe both the problem (requirements) and the solution (design) in order to gain a better understanding of the issues involved. Once a model has been constructed it can be analyzed to uncover flaws and expose fundamental issues [23]. This role of models cannot possibly be assumed by code. The idea is not new, but there is a recent trend towards more use of models in mainstream circles of software engineering. This is the goal of MDSE [19], which tries to alleviate the complexity of software development by using models. Model transformation has a key role in MDSE. A model transformation takes as input a model conforming to a given meta-model and produces as output another model conforming to a given meta-model. One of the characteristics of a model transformation is that it is also a model, i.e. it conforms to a given meta-model.

There are two reasons for against-our-expectation behavior of the software [25]: either there are shortcomings or omissions in the original specification, or the software does not conform to its specification. These two issues result from the following causes: 1) incomplete, ambiguous, and inconsistent requirements specification, 2) imprecise and imperfect verification of the specification and design which in turn lead to incomplete and untimely discovery of the software's errors during the development cycle. These problems arise from the weaknesses of informal and semi-formal modeling methods (SFMMs) in specification and verification of the software requirements.

This paper investigates the advantages and shortcomings of SFMMs and formal modeling methods (FMMs) by surveying the literature [1][5][13][25]. Reference [26] has already investigated the advantages and disadvantages of SFMMs and FMMs, empirically, by specifying the multi-lift system case study. The most important conclusion is that each modeling method has some unique advantages and limitations. Using only one of them as the sole approach leads not to satisfy all required aspects of software quality such as reliability, flexibility, reusability, scalability, and so on [30]. Combination of these methods is necessary to successfully understand, analyze, specify, validate, and verify requirements, problems, and solutions. Although, there are several valuable attempts to integrate these methods to utilize unique advantages of both formal and semi-formal modeling methods, there is a long way ahead to achieve the promised goals.

This paper proposes a new approach to enhance the software development process. This work emphasizes on the software behavior rather than its structure. In the proposed approach, the formalism plays the key role, i.e., the structure and behavior of the software is initially modeled using a suitable formal modeling language (such as Object-Z). These formal models, along with formal refinement [3] ensure correctness and reliability. Then, with an iterative and evolutionary approach and in specific intervals, software behavior is extracted from formal models to be visualized in a semi-formal modeling language (such as UML). Visualized behavior increases and facilitates the interactions among project stakeholders (such as analyzers and designers), who are not, necessarily, familiar enough with complex mathematical concepts of formal methods. This also provides the possibility of applying design patterns on visualized behavior to improve its flexibility, reusability, and scalability. So, potential shortcomings and inconsistencies of the software behavior are identified and, consequently, required changes are applied and a newly improved version of the formal behavior is produced. The improved models are then re-formalized. The proposed approach is a step towards development of correct, reliable [6], flexible, reusable, and scalable software through enabling the construction of formal models from semi-formal ones (formalizing) and vice versa (visualization) during an iterative and evolutionary approach. References [26] and [27] present a case study in order to show the proposal applicability.

A detailed study regarding visualization and formalization is given in [1]. All related works are just a step in the right direction, but much more is yet to be done. The most frequently adopted approach is to define transformations between the visual and formal models [1][2][4][7][11][12][14][18][20][23][24]. However, a significant problem with these suggested approaches is that the transformation itself is often described imprecisely, with the result that the overall transformation task may be imprecise and incomplete. Consequently, the confidence the developer may have in the models is reduced, making the transformation approach unreliable.

The rest of this paper is organized as follows: Section 2 presents the motivation of the work by describing the reasons of integrating SFMMs and FMMs and its importance. The advantages and limitations of semi-formal and formal modeling methods are also investigated according to the literature review in this section. Section 3 defines the problem to be solved by the proposed approach. Finally, Section 4 discusses future work and draws conclusions.

II. MOTIVATION

This section describes the motivation of this paper via elaborating the benefits and limitations of SFMMs and FMMs according to the literature review.

A. Semi-formal Modeling Methods

SFMMs consist of a development method and a collection of notations for modeling software systems. UML is a unification of semi-formal modeling notations [23][31]. In summary, the main strengths of semi-formal techniques are as follows:

- Semi-formal notations are graphical, making them appealing, intuitive, and easy to be adopted. They are good at describing particular aspects of systems, abstracting away from details, and giving a good overall picture of what is being described. Sometimes they do not require a great deal of expertise to be understood. So they provide a good medium for discussions with clients.
- SFMMs are more than just a notation. They provide step-by-step guidance on how to approach problems. They encourage problem decomposition, which helps to reduce complexity.

Lack of a sound mathematical basis is the major weakness of SFMMs. They do not have a formal semantics. There are several problems related to their semantics:

- Either they are defined informally and vaguely using natural language, or they are defined through meta-modeling using some meta-language that is not precisely defined.
- Developers tailor the interpretation of diagrams to the problem at hand informally, tacitly, and sometimes unconsciously. This constitutes a source of confusion and ambiguity. Such misinterpretations might be even greater if the specification volume is large or development team crosses national and cultural boundaries [5].

These limitations lead to lack of means for mechanical analysis. They can also make the understanding more apparent than real; All is too easy and superficial, and the specifier is never confronted with the relevant issues. As a result, semi-formal methods cannot produce a precise, complete, and consistent specification. Specification plays a vital role in producing reliable software. Design and subsequent implementation is based upon the specification. Misunderstandings in the specification lead to the delivery of final applications that do not match user requirements. Moreover, testing is always carried out with respect to requirements as laid down in the specification. If the specification document is in any way ambiguous it is open to interpretation, and hence misinterpretation, making testing a rather inexact science.

Next section shows how the formal methods help in covering the weaknesses of SFMMs in specification, validation, and verification.

B. *Formal Modeling Methods*

FMMs are inspired by the way mature engineering disciplines build their artifacts: based on prediction and calculation with sound mathematical theories. Formal methods are utilized in all phases of software development process. FMMs, using formal languages such as Object-Z [7], provide the software with a precise, unambiguous, and abstract specification. In the next steps, required details are added to the initial abstract specification through an evolutionary process, including some design steps towards the final program. Accordingly, the initial formal specification is gradually refined. The refinement process will proceed until the generation of the final code [3]. Certain notations of formal methods support the notion of formal refinement. Formal refinement ensures that these refinements and transformations are correct. The correctness of a refinement is demonstrated through mathematical proof [23]. The benefits of using the formal modeling techniques have been recognized as follow:

- Formal modeling helps to gain a deep understanding of the system and its domain. It encourages the specifier to be abstract, yet rigorous and precise, forcing the modeler to ask all sorts of questions.
- Formal modeling clarifies the customer's vague ideas, revealing ambiguities, inconsistencies, and incompleteness in the requirements [23].
- The analysis of formal models can be used to support verification and validation. In verification, a formal model can be proved or checked for the satisfaction of desired properties, and that a refined design or implementation satisfies its specification. In validation, a requirements model can be checked against its requirements for white-box system testing either through animation or proof, and for black-box system testing by generating test cases from the model.

Although the increased rigor, precision and means of calculation that formal techniques offer seems indisputable [22], formal methods have not been taken up by industry. To explain this, many reasons have been hypothesized, education being one of them. So, FMMs have been embraced only in domains where reliability is absolutely crucial, such as safety-critical, security-critical, and high integrity systems [5]. Some other recognized shortcomings of FMMs are given below:

- Formal methods are notorious for being hard. Substantial efforts are required for formal modeling and verification. They are only effectively usable by highly-skilled experts.
- Most formal methods are suited to describe particular aspects of systems, but usually not all aspects. The problem occurs when all aspects need to be modeled.
- Formal methods provide a notation to write models and approaches to analyze them. However, software engineering practices require further support: guidelines, approaches to modeling, and patterns.
- The large variety of formal methods makes the choice of a particular one difficult.
- Most formal methods have little automated support beyond type-checking; developers are usually left the onus of performing proofs, which demand too much time and expertise for practical application.
- Practitioners need to be trained, and, since there is not much experience in using formal methods, the costs associated with their use are high. They also require an investment of time and money in specification, before any code is written.

The main conclusion is that FMMs and SFMMs have some advantages and limitations. Using only one of them as the sole approach leads not to satisfy all required aspects of software quality. This paper advocates an approach to building a framework for rigorous MDSE based on combining UML as a semi-formal language with Object-Z as a formal modeling language. SFMMs are supplemented with FMMs to introduce rigor in the development and to sweeten formal methods usage with diagrams.

III. Problem Definition

The problem to be investigated by this work is defined in this section. Solving this problem is a step towards developing high quality software. To do so, a new approach based on integrating Object-Z, as a formal, and UML, as a semi-formal modeling language, is proposed.

Using FMMs as the sole approach to software development leads to reliable software but with the following issues:

1. There are different interpretations of the initial informal requirements by customer and development team. There is also possibility of changing requirements during software development. These issues end to production of a software in contrary with the initial requirements. Fig. 1 illustrates this problem. There are two reasons for such an incorrect result: 1) there is no possibility of proving a perfect match between actual informal requirements and initial formal specification (δT_1), 2) it is difficult to do validation in the interval δT_2 because of the trouble in understanding the formal models. So formal methods, certainly brings us to a result that conforms to the initial formal specification (because of formal refinements), however, it does not necessarily conform to the actual informal requirements.

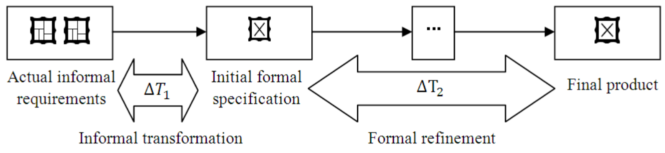

Figure 1. Imprecise interpretation of customer requirements

Visualization is an approach to solve the first problem, which leads to facilitate requirements validation in the interval δT_2 [15]. However, prototyping [16] is a better solution for requirements validation. To do so, the formal specification should be transformed so that its new form can be executed or animated [16][32].

2. Even assuming that the initial formal specification exactly represents the actual informal functional requirements of the customer, we still do not reach the software with good enough quality of non-functional requirements such as reusability, flexibility, scalability, and extendibility. There are two reasons for such an unexpected result: 1) difficulty in utilizing the heuristic and narrative techniques of software engineering such as design patterns in the interval δT_2 , 2) inability of development team members such as analyzers and designers in understanding complex mathematical concepts of formal languages.

This work aims to solve the second problem. To do so, a new approach is suggested to improve software development process by combining Object-Z and UML to achieve high quality models of specification and design. In other words, this work proposes a new approach to develop high quality software through model transformation between Object-Z and UML. Fig. 2 illustrates a schematic view of the new proposed approach. Visualization facilitates understanding of the formal models and subsequently provides possibility of interaction with stakeholders, who are not necessarily familiar enough with complex mathematical concepts of formalism. It also simplifies using the narrative techniques of software engineering such as design patterns during software development process.

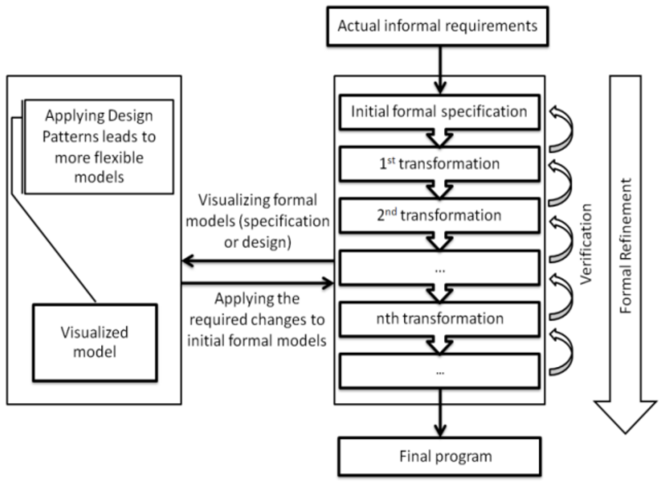
Actual informal requirements
Applying Design Patterns leads to more flexible models
Initial formal specification
1st transformation
2nd transformation
...
nth transformation
...
Visualizing formal models (specification or design)
Applying the required changes to initial formal models
Visualized model
Verification
Formal Refinement
Final program

Figure 2. A schematic view of the proposed approach

As illustrated in Fig. 2, the *initial formal specification* is produced as the first artifact, according to the informal requirements of the stakeholders, using Object-Z. The initial formal specification is then refined using several transformations. Details of design are gradually added to the initial formal specification during transformations referred to as formal refinement. Formal refinement ensures correctness and reliability of the produced artifacts. In time of reviewing the artifacts from the aspect of behavioral design patterns, the last refined formal artifact is visualized in a dominant semi-formal modeling language, i.e., UML. UML diagrams make it possible to revise the structure and behavior of the software from the view points of design patterns. The visualized model is then gradually revised using behavioral design patterns. Such a revision improves the flexibility and reusability of the visual models. The last revised visual model is then re-formalized in Object-Z. Repeatedly, the more required details of design or even implantation are augmented to the formal model using formal refinement. Such an iterative and evolutionary process continues until achieving a final product with the desired quality.

Software includes two aspects: structure (static) and behavior (dynamic) [16][21]. The proposed approach concentrates on software behavior. It facilitates analyzing and validating the behavioral aspect of formal models of software by visualization. Visualization prepares an appropriate ground to use heuristic and narrative principles of software engineering such as behavioral design patterns during software development process. So, the potential shortcomings and inconsistencies of the behavioral aspect of these models are identified. This improves the process of gradual augmentation of design decisions to the initial formal specification. Such an improvement leads to more flexibility, reusability, and scalability in developing software.

Design patterns are high level building blocks that promote elegance in software by ordering proven and timeless solutions to common problems in software design. Applying design patterns in software design has important effects on software quality metrics such as flexibility, reusability, scalability, and robustness [9][22][28][29][33]. There are three types of design patterns, including structural, creational, and behavioral patterns [8][9]. According to the above-mentioned goal of this work, we focus on the behavioral patterns (such as mediator, observer, and state) which shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Object-oriented design encourages the distribution of behavior among objects to increase software reusability and flexibility. An important issue here is how peer objects know about each other. Peers could maintain explicit references to each other, but that would increase their coupling. Though distributing software into many objects generally enhances reusability and flexibility, proliferating interconnections tend to reduce reusability again. Moreover, it can be difficult to change the software behavior in any significant way, since behavior is distributed among many objects. Such a difficulty decreases the flexibility again. As a result, you may be forced to define many subclasses to customize the software behavior. The mediator pattern avoids this by introducing a mediator object between peers. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. In this respect, we attempt to propose a systematic approach to improve the quality of formal design from the viewpoint of the mediator design pattern. That is, a formal design, in Object-Z, is received as an input, and then behavior of this formal design is abstractly visualized, in UML, as an output. Indeed, there is a focus on visualizing those aspects of the software behavior that are prone to revising from the viewpoint of the mediator pattern. Moreover, this approach, after full implementation, will automatically explore and recognize the suitable times in order to review the software behavior from the view point of mediator pattern throughout the software development process.

Moreover, software distribution into a collection of cooperating classes requires maintaining consistency among related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability and flexibility. Observer pattern define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In short, the required activities to visualizing the software behavior (by focus on those aspects of behavior that are required for revision from the viewpoint of observer pattern) include: 1) systematic elicitation of the objects that their states are dependent on each other, 2) visualizing the discovered objects as appropriate candidates for review, as well as 3) automatic proposing of the suitable times to review the software behavior from the viewpoint of observer design pattern.

Strategy pattern define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. We use the strategy pattern when: l) many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors, 2) you need different variants of an algorithm. Strategies can be used when these variants are implemented as a class hierarchy of algorithms, and 3) a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class. So the Strategy pattern increases the flexibility through defining families of related algorithms, preventing subclassing, and eliminating conditional statements. Summarily, the required activities to visualize the software behavior form the viewpoint of strategy pattern include: 1) systematic discovery and elicitation of the classes that have several behaviors, 2) visualizing the discovered classes as appropriate candidates for review, as well as 3) automatic proposing of suitable times for software behavior review from the viewpoint of the strategy design pattern.

In all above-mentioned revision processes, the required changes, revealed after visualization, are re-formalized and thus the primary formal models are improved from the view point of behavioral design patterns. Software behavior is visualized from the required aspects using the suitable diagrams of UML such as class diagram [15][16]. Class diagram makes it possible to revise the structure and behavior of the software from the view points of design patterns

There has been an evolution in the way of transforming the models [10][17]. In model transformation, the most important issue is how to preserve the semantic and the syntactic structure of model elements. To do so, this work tends to propose a formal bidirectional meta-model-based transformation between UML and Object-Z. To do so, a meta-model should be formally defined for Object-Z in a similar architecture to which the UML meta-model is defined [11]. Then these meta-models will be used to define a systematic transformation between the two languages at the meta-level. In this way, we can provide a precise, consistent, and complete transformation between the two languages preserving the semantics and the syntactic structure of models presented in both languages. Since UML and Object-Z share basic object-oriented concepts, an attempt to create a systematic transformation between the two languages seems sound. Proposing such a meta-model-based mechanism is left for future work. In the following subsections, as an instance, we show how a common construct between UML and Object-Z such as class can be formally defined at the meta-level in a unified format using Object-Z [11]. Then a formal rule is presented to transform class construct from UML to Object-Z based on the formal definitions of class in UML and Object-Z at meta-level.

A. Formal definition of UML class

A UML class has a name, attributes, and operations. An attribute has a name, a visibility, a type, and a multiplicity. An operation has a name, a visibility, and parameters. Each parameter of an operation has a name and a given type. Prior to formalizing classes, we define a given set, *Name,* from which the names of all classes, attributes, operations, operation parameters, associations, and roles are drawn:

$[Name]$

The class *UMLType*, as an Object-Z class, is a meta-type, from which all possible types in UML such as object types, basic types (integer and string), and so on can be derived. Each type has a name and contains a collection of its own features: attributes and operations. Thus, a circled c which models a containment relationship in Object-Z is attached to the types of *attributes* and *operations*.

$UMLType$

$name : Name$
$attributes : \mathbb{F}\ UMLAttribute_{\copyright}$
$operations : \mathbb{F}\ UMLOperation_{\copyright}$

Attributes and parameters are also defined as follows. Variable *multiplicity* in *UMLAttribute* describes the possible number of data values for the attribute that may be held by an instance. Visibility in UML can be private, public, or protected.

$$VisibilityKind ::= private \mid public \mid protected$$

UMLAttribute

$$\begin{array}{l} name : Name \\ type : \downarrow UMLType \\ visibility : VisibilityKind \\ multiplicity : \mathbb{P}_1 \mathbb{N} \end{array}$$

UMLParameter

$$\begin{array}{l} name : Name \\ type : \downarrow UMLType \end{array}$$

Within an operation, parameter names should be unique.

UMLOperation

$$\begin{array}{l} name : Name \\ visibility : VisibilityKind \\ parameters : \mathrm{seq}\, UMLParameter_{©} \\ \hline \forall\, p1, p2 : \mathrm{ran}\, parameters \bullet p1.name = p2.name \Rightarrow p1 = p2 \end{array}$$

With these classes, an Object-Z class *UMLClass* is defined as follows. Since a class is a type, it inherits from *UMLType*. Attribute names defined in a class should be different and operations should have different signatures. The class invariant formalizes these properties.

UMLClass

UMLType

$$\begin{array}{l} \forall\, a1, a2 : attributes \bullet a1.name = a2.name \Rightarrow a1 = a2 \\ \forall\, op1, op2 : operations \bullet \\ \quad (op1.name = op2.name \land \#op1.parameters = \#op2.parameters \land \\ \quad \forall\, i : 1 \mathrel{..} \#op1.parameters \bullet \\ \qquad op1.parameters(i).name = op2.parameters(i).name \land \\ \qquad op1.parameters(i).type = op2.parameters(i).type) \Rightarrow op1 = op2 \end{array}$$

B. Formal definition of Object-Z class

First, the semantics of type *Name* is extended to include the names of all classe, attributes, operations, and operation parameters in Object-Z. The following Object-Z class *OZType* is a formal description of metaclass *OZType*. In the metamodel, *OZType* is an abstract class from which all possible types in Object-Z can be derived.

OZType

$$name : Name$$

The Object-Z class *OZAttribute* is a formal description of attributes. Each attributs has a name, a type, and a multipilicity constraining the number of values that the attribute may hold. It also has an attribute, *relationship*, to represent whether this attribute models a relationship between objects. Like UML, relationships between objects can be common reference relationships, shared, or unshared containment relationships. For this, we define an enumeration type, *RelationshipKind*, which can have *relNone*, *reference*, *sharedContainment*, and *unsharedContainment* as its values. The value *relNone* represents pure attributes of a class. When an attribute models a relationship, the attribute *navigability* represents the direction of the relationship (although the navigability of a relationship is modeled impilicitly in Object-Z). Visibility in Object-Z can be public or private.

$$VisibilityKind ::= private \mid public$$

$$\begin{array}{l} RelationshipKind ::= relNone \mid reference \mid \\ \qquad\qquad sharedContainment \mid unsharedContainment \end{array}$$

$$NavigabilityKind ::= navNone \mid bi \mid one$$

OZAttribute

$$\begin{array}{l} name : Name \\ type : \downarrow OZType \\ visibility : VisibilityKind \\ multiplicity : \mathbb{P}_1 \mathbb{N} \\ relationship : RelationshipKind \\ navigability : NavigabilityKind \end{array}$$

We formalize *OZParameter* and *OZOperation* in the same way as *OZAttribute*.

OZParameter

$$\begin{array}{l} name : Name \\ type : \downarrow OZType \end{array}$$

OZOperation

$$\begin{array}{l} name : Name \\ visibility : VisibilityKind \\ parameters : \mathrm{seq}\, OZParameter_{©} \end{array}$$

Now we are in the position to formalize Object-Z classes. An Object-Z class named *OZClass* is a formal description for classes in Object-Z. Since classes are a kind of type, *OZClass* inherits from *OZType*. The attribute *superclass* maintains inheritance information of classes. Each class has its own attributes and operations defining static and dynamic behaviors of its instances. Circular inheritance is not allowed. Attribute and operation names should be unique within a class. These properties are specified in the predicate of *OZClass*. Functions *directSuperclass* and *allSuperclass* return direct superclass of a class and all inherited *superclasses* of a class, respectively.

$$
\begin{array}{|l}
directSuperclass : OZClass \rightarrow \mathbb{P}\, OZClass \\
allSuperclass : OZClass \rightarrow \mathbb{P}\, OZClass \\
\hline
\forall\, oc : OZClass \bullet \\
\quad directSuperclass(oc) = oc.superclass \\
\quad allSuperclass(oc) = directSuperclass(oc) \cup \\
\quad (\bigcup \{sco : directSuperclass(oc) \bullet allSuperclass(sco)\})
\end{array}
$$

$$
\begin{array}{|l}
\hline
OZClass \\
OZType \\
\begin{array}{|l|}
\hline
superclass : \mathbb{F}\, OZClass \\
attributes : \mathbb{F}\, OZAttribute_{\copyright} \\
operations : \mathbb{F}\, OZOperation_{\copyright} \\
\hline
self \notin allSuperclass(self) \\
\forall\, a1, a2 : attributes \bullet a1.name = a2.name \Rightarrow a1 = a2 \\
\forall\, op1, op2 : operations \bullet op1.name = op2.name \Rightarrow op1 = op2 \\
\hline
\end{array} \\
\hline
\end{array}
$$

C. *Formal transformation rule for class*

As illustrated in Fig. 3, a formal description for mapping a UML class to an Object-Z class is given by function *mapUMLClassToOZ* that takes a UML class and returns the corresponding Object-Z class. The UML class name is used as the Object-Z class name. All attributes of the UML class are declared as attributes in the state schema of the corresponding Object-Z class. Also, each operation in the UML class is translated to an operation schema. In UML, types of attributes are a language-dependent specification of the implementation types and may be suppressed. Types of attributes in Object-Z are language-independent specification types and cannot be omitted. Operations parameters are similar. Detailed transformation rules regarding attribute types and operation parameter types are not provided. Instead, an abstract function, *convType* is defined that maps a UML type to an Object-Z type. Visibility and multiplicity features are mapped to those of Object-Z.

An appropriate evaluation method helps determine the overall effects of the new approach in relation to promised objectives. This method also includes any recommendations for improvement. As previously mentioned, the major goal of introducing the new approach is to improve the process of formal modeling (including specification and design) of software behavior based on visualization. So we should measure the capability of the suggested approach in satisfying the expected goals. Evaluation criteria of the proposed approach include: 1) correspondence percentage between visual and formal models transformed to each other by the proposed meta-model based transformation method, 2) the amount of increasing the quality (such as flexibility, reusability, and scalability) of the developed software using the proposed method. As we intend to propose a meta-model-based transformation approach, a formal and systematic transformation between the two languages will be defined at the meta-level. So we can prove the correctness, precision, and completeness of the transformation mathematically. In addition, to demonstrate the proposed approach, a high quality multi-lift system as a non-trivial case study will be developed using the proposed approach.

IV. CONCLUSION AND FUTURE WORK

Although, the widespread use of SFMMs in mainstream software development provides the possibility of developing flexible, reusable, and scalable software, it does not lead to software reliable enough for safety-critical purposes. Their semantics are not well defined. FMMs have precise semantics, allowing for unambiguous models of systems to be specified and designed. However, their use has not been widely adopted due to the mathematical nature of the languages.

$$
\begin{array}{|l}
convType : \downarrow UMLType \rightarrowtail \downarrow OZType
\end{array}
$$

$$
\begin{array}{|l}
mapUMLClassToOZ : UMLClass \rightarrow \mathbb{P}\, OZClass \\
\hline
\forall\, uc : UMLClass \bullet \\
\quad mapUMLClassToOZ(uc) = \{oc : OZClass \mid\ uc.name = oc.name \wedge \\
\quad\quad \forall\, ua : uc.attributes \bullet \\
\quad\quad\quad \exists\, oa : oc.attributes \bullet \\
\quad\quad\quad\quad oa.name = ua.name \wedge\ oa.type = convType(ua.type) \wedge \\
\quad\quad\quad\quad oa.visibility = ua.visibility \wedge\ oa.multiplicity = ua.multiplicity \wedge \\
\quad\quad\quad\quad oa.relationship = relNone \wedge\ oa.navigability = navNone \\
\quad\quad\quad \forall\, uo : uc.operations \bullet \\
\quad\quad\quad\quad \exists\, oo : oc.operations \bullet \\
\quad\quad\quad\quad oo.name = uo.name \wedge\ oo.visibility = uo.visibility \\
\quad\quad\quad\quad \forall\, up : \operatorname{ran} uo.parameters \bullet \\
\quad\quad\quad\quad\quad \exists\, op : \operatorname{ran} oo.parameters \bullet \\
\quad\quad\quad\quad\quad\quad op.name = up.name \wedge\ op.type = convType(up.type)\}
\end{array}
$$

Figure 3. Formal transformation rule for class

Investigation of integrated methods has taught us many things: (a) visual modeling notations and formal methods can coexist within the same development and complement each other when developing software models, (b) this coexistence is useful and provides many benefits, and (c) formalization of diagrammatic languages, like UML, and visualization of formal models, like Object-Z, is far from trivial.

This work proposes a new approach for integrating visual and formal models to ensure achieving more flexible, reusable, scalable, yet reliable software. To do so, we propose a precise mechanism to transform graphical models into formal specifications and vice versa. This work intends to present a meta-model-based transformation between UML and Object-Z. The two languages will be defined in terms of their meta-models, and a systematic transformation between the models will be provided at the meta-level. As a result, we provide a precise, consistent, and complete transformation between visual models in UML and formal models in Object-Z. Visualizing the formal models of the software behavior prepares an appropriate ground to revise them from the viewpoints of design patterns. Although, this paper draws the path towards solving the defined problem and achieving the promised goals, proposing the meta-model-based transformation is left for future work.

REFERENCES

[1] N. Amálio, Generative frameworks for rigorous model-driven development, PhD thesis, Dept. of Computer Science, University of York, 2006.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A Challenging Model Transformation", Proc. ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS), pp. 436-450, 2007.

[3] D. Bjørner, Software Engineering 3: Domains, Requirements, and Software Design, Springer, 2006.

[4] F. Bouquet, F. Dadeau, and J. Groslambert, "Checking JML specifications with B machines", Proc. ZB 2005, LNCS, vol. 3455, Springer, pp. 434-453, 2005.

[5] Q. Charatan and A. Kans, Formal Software Development: From VDM to Java, Palgrave Macmillan, 2004.

[6] R. N. Charette, "Why software fails", IEEE Spectrum, vol. 42(9), pp. 42-49, 2005.

[7] R. Duke and G. Rose, Formal Object-Oriented Specification Using Object-Z, MacMillan Press, 2000.

[8] E. Freeman, E. Freeman, and B. Kathy Sierra, Head First Design Patterns. O'Reilly Media, First edition, 2004.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Pattern: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, Fifth printing, 1995.

[10] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, "Generating Transformation Rules from Examples for Behavioral Models", Proc. Second International Workshop on Behavior Modeling: Foundation and Applications, Paris, France, 2010.

[11] S. Kim and D. Carrington, "A formal meta-modeling approach to a transformation between the UML state machine and Object-Z", Proc. ICFEM 2002, LNCS, vol. 2495, Springer, pp. 548-560, 2002.

[12] H. Miao, L. Liu, and L. Li, "Formalizing UML models with Object-Z", Proc. ICFEM2002, Springer-Verlag, pp. 523-534, 2002.

[13] J. Bowen and M. Hinchey, "Seven more myths of formal methods", IEEE Software, vol. 12 (4), pp. 34-41, 1995.

[14] I. Poernomo, "Proofs-as-model-transformations" LNCS, vol. 5063, pp. 214-228, 2008.

[15] F. Polack, "SAZ: SSADM version 4 and Z", Proc. Software Specification Methods: an overview using a case study, Springer, pp. 21-38, 2001.

[16] I. Porres, "Modeling and Analyzing Software Behavior in UML", PhD thesis, Department of Computer Science, Abo Akademi University, Finland, 2001.

[17] R. Pressman, Software Engineering: A Practitioner's Approach, 7th edition, McGraw Hill, 2009.

[18] S. K. Rahimi, "Specification of UML Model Transformations", Proc. Third International Conference on Software Testing, Verification and Validation, pp. 323-326, Paris, 2010.

[19] R. Razali, C. Snook, M. Poppleton, and P. Garratt, "Usability Assessment of a UML-based Formal Modeling Method Using Cognitive Dimensions Framework", Human Technology, 2008.

[20] D. C. Schmidt, "Model-driven engineering", IEEE Computer, 39 (2), pp. 25-31, 2006.

[21] C. Snook and M. Butler, "UML-B: Formal modeling and design aided by UML", ACM Trans. Softw. Eng. Methodol, vol. 15 (1), pp. 92-122, 2006.

[22] I. Sommerville, Software Engineering, 8th edition, Addison Wesley, June 4, 2006.

[23] S. Stepney, F. Polack, and I. Toyn, "Patterns to guide practical refactoring: examples targeting promotion in Z", Proc. ZB 2003, Finland, LNCS, vol. 2651 of, Springer, pp. 20-39, 2003.

[24] J. R. Williams, Automatic Formalization of UML to Z, MSc Thesis, Department of Computer Science, University of York, 2009.

[25] J. Ludewig, "Models in software engineering – an introduction", Software and Systems Modeling, vol. 2(1), Springer-Verlag, 2003.

[26] A. Rasoolzadegan and A. Abdollahzadeh, Specifying a Parallel, Distributed, Real-Time, and Embedded System: Multi-Lift System Case Study, Technical Report, Information Technology and Computer Engineering Faculty, Amirkabir University of Technology, Tehran, Iran, 2011: http://ceit.aut.ac.ir/~86131901/Publications.htm.

[27] A. Rasoolzadegan, A. Abdollahzadeh, "Empirical Evaluation of Modeling Languages Using Multi-Lift System Case Study", Proc. MSV'11: The 8th annual International Conference on Modeling, Simulation and Visualization Methods, Las Vegas, Nevada, USA, 2011.

[28] S. Blazy, F. Gervais, and R. Laleau, "Reuse of specification patterns with the B method". Proc. ZB 2003, Turku, Finland, LNCS, vol. 2651, Springer, pp. 40-57, 2003.

[29] A. Flores, R. Moore, and L. Reynoso, "A formal model of object-oriented design and GoF design patterns", Proc. FME 2001, LNCS, vol. 2021, pp. 223-241, Springer, 2001.

[30] A. H. Eden and T. Mens, "Measuring Software Flexibility", IEE Software, vol. 153(3), pp. 113-126. London, UK: The Institution of Engineering and Technology, 2006.

[31] OMG, Object Constraint Language (OCL). version 2.0, Object Management Group, 2006: http://www.uml.org.

[32] H. Liang, J. Song Dong, J. Sun, and W. Wong, "Software monitoring through formal specification animation", Innovations in Systems and Software Eng., vol. 5(4), pp. 231-241, 2009.

[33] S. Kim and D. Carrington, "A rigorous foundation for pattern-based design models", Proc. ZB 2005, LNCS, vol. 3455, Springer, pp. 242-261, 2005.