A New Approach to Reliable, yet Flexible Software

Abbas Rasoolzadegan

Intelligent Systems Laboratory (http://ce.aut.ac.ir/islab)
Information Technology and Computer Engineering Faculty
Amirkabir University of Technology (Tehran Polytechnic)
rasoolzadegan@aut.ac.ir

Supervised by Professor Ahmad Abdollahzadeh Barforoush

Abstract. Developing reliable, yet flexible software is a hard problem. Formal methods take a precise approach to software development, delivering reliable software; however, in addition to high cost involvements, they require a level of expertise that is not common in commercial development communities. These limitations lead to decreasing their practicality. Semi-formal methods, which are widely used in practical large-scale software development, do not take a rigorous approach to reliability of software in development. Investigation of advantages and limitations of semi-formal and formal methods, theoretically (by surveying the literature) and empirically (by defining a suitable case study), shows that combination of both methods ensures achieving high quality models which in turn lead to flexibility and more reliable software. This work proposes a new approach to integrate formal (Object-Z) and semi-formal (UML) notations using a bidirectional, precise, and consistent meta-model-based transformation. Accordingly, software is initially modeled using Object-Z. These formal models, along with formal refinement ensure reliability. With an iterative and evolutionary approach, formal models are visualized in UML. Applying design patterns on visualized models improves flexibility. The improved models are then re-formalized.

Keywords: UML, Object-Z, design patterns, meta-model-based transformation, model transformation.

1 Introduction

Software permeates our daily life. There is probably no other human-made material which is more omnipresent than software in our modern society. It has become a crucial part of many aspects of society. Size and complexity of software systems have grown dramatically during the past few decades, and the trend will certainly continue in the future. Because of this ever-increasing dependency, software failures can lead to serious, even fatal, consequences in safety-critical systems as well as in normal business [7]. Moreover, rapid technological developments pervade every aspect of daily life, having direct effect on the software we use. Every element of the software's operational environment is in a state of constant flux [32]. Contemporary literature recognizes the central role of *reliability* and *flexibility* in software development.

Studies show that the major causes of most software failure are imprecise and incomplete understanding of customer requirements as well as lack of precise, careful, and comprehensive elicitation, specification, analysis, validation, and verification of them during *Requirements Engineering* (RE) in software development cycle [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 the design of its solution.

The increasing importance of developing reliable and flexible software beside increasing necessity of RE as well as need for further abstraction lead to increasing use of various types of *models* [3], [31]. Models are used at different phases of software development, ranging from requirements to detailed design for specification,

analysis, validation, and verification of customer requirements, problems and also design of their solutions. This is the idea behind *Model-Driven Software Engineering* (MDSE) [27], an approach that advocates models, rather than code, as the primary artifacts of software development. *Model transformation* and *model refinement* have key roles in MDSE. Model transformation includes *model-to-text* transformation to generate code from models, *text-to-model* transformations to parse textual representations to model representations, *model extraction* to derive higher-level models from legacy code, and *model-to-model* transformations to normalize, weave, optimize, and refactor models, as well as to translate between modeling languages. The fact that models may have different levels of abstraction 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.

Semi-formal methods (SFMs) using semi-formal languages have a visual nature, and it is this, together with their pragmatic approach to development, that explains their popularity. They have emerged from a need to abstract away from the details of code and to visualize the overall system structure and behavior. The main strengths of semi-formal techniques include: intuitive and widely known notations and methodological support emphasizing problem decomposition. Lack of a sound mathematical basis is the major weakness of SFMs which in turn leads to different interpretations [6], [19] as well as ambiguous, imprecise, and inconsistent specification. Misunderstandings in the specification lead to against-user-expectation behavior of the software. SFMs also suffer from lack of means for mechanical analysis.

Formal methods (FMs) using formal languages provide the software with precise, unambiguous, and abstract specification during development cycle. In the next design steps, required details are gradually added to initial specification through an evolutionary formal refinement process [3], [31]. FMs are based on prediction and calculation with sound mathematical theories. They are utilized throughout software development life cycle [13] for precise specification, complete analysis, and rigorous verification and validation of software requirements. However, they require expertise. So, customers, users, analyzers, and designers (who do not, necessarily, master enough their complex mathematical concepts) are unable to analyze and validate the formal models. Therefore, FMs have not been taken up by industry. They have been embraced only in domains where reliability is absolutely crucial, such as safety-critical, security-critical, and high integrity systems [6] because of their high up-front cost. Other kinds of systems, however, are driven by other priorities, such as time to market, feature count, and cost of production. The problem with formal methods, as they stand today, is that they clearly conflict with such priorities. They also suffer from lack of methodological and tool supports.

We conclude that FMs and SFMs have some unique advantages and limitations. Using only one of them as the sole approach leads not to reliable and flexible software but to reliable or flexible software. So, we aim to supplement the semi-formal methods with formal ones in order to introduce rigor in the development, and to sweeten formal methods usage with visualization. Such an integration results in developing software with the desired quality.

We have taken a non-trivial system, namely the multi-lift system as a case study [34]. This system is a commonly used test bed to demonstrate the expressive power of various modeling languages in specifying concurrent reactive systems. It is not a trivial case study because of the complexity caused by inherent concurrent interaction in the system.

Investigation of the advantages and limitations of semi-formal and formal methods, theoretically (by surveying the literature) and empirically (by defining a suitable case study) shows that combination of both methods would be necessary for precise and complete requirements specification, validation, and verification as well as flexible

and reliable design in a practical approach. Such a combination ensures achieving high quality models. More information about the details of the investigation performed on the advantages and limitations of FMs and SFMs according to the literature review and empirical experiments is publically accessible via http://ceit.aut.ac.ir/islab/Researches/RE/Appendix.doc.

We strongly believe that it is possible to integrate FMs into mainstream software modeling practice. This can be done in a flexible and practical way, so that the integration has an engineering value and is actually worth doing. There have been many approaches that perform such integrations with very positive results. So, before the work reported here even started, there was already strong evidence in favor of its possibility. There are also different ways to achieve the goal of practicality [1]. So, we aim to follow a rigorous approach to MDSE that is practical based on combining semi-formal notations with formal modeling languages regarding to their advantages and limitations. The rigor will be given by the use of formal models. The practicality will be provided through application of semi-formal models which are accessible to a wide range of developers who do not need to be formal methods experts. In fact, this paper enables the construction of formal models from diagrams (*formalizing*) and vice versa (*visualization*). To do so, this work proposes a precise, consistent, and complete transformation between visual models in UML and formal models in Object-Z.

This work emphasizes on the software behavior rather than its structure. In the proposed approach, the formalism plays the key role, i.e., the software is initially modeled using Object-Z. These formal models, along with formal refinement ensure reliability. Then, with an iterative and evolutionary approach and in specific intervals, software behavior is extracted from formal models to be visualized in 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. So potential errors 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 practical step towards development of correct, reliable, and flexible software.

The rest of this paper is organized as follows: Section 2 defines the problem that will be solved by the proposed approach. The goals and outputs of this work are also expressed in this section. In section 3 the problem solving method is described, i.e. the path towards solving the defined problem and achieving the promised goals is drawn. Testing and evaluation method of the new approach is then presented. Finally, Section 4 discusses future work and draws some conclusions.

2 Problem Definition

The problem to be investigated by this work is defined in this section. Solving this problem is a step towards developing reliable, yet flexible software. To do so, a new approach based on integrating Object-Z and UML 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. Figure 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.

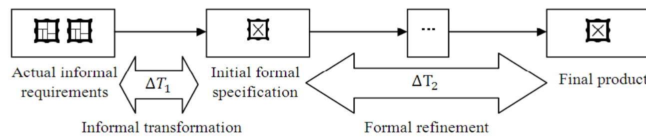

Figure1. Incorrect and imprecise interpretation of customer requirements

Visualization is an approach to solve this issue which leads to facilitate requirements validation in the interval ΔT_2 [23]. However, prototyping [24] 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 [17], [24].

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 semi-formal 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 this 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 which in turn lead to more reliable and flexible software. Figure 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.

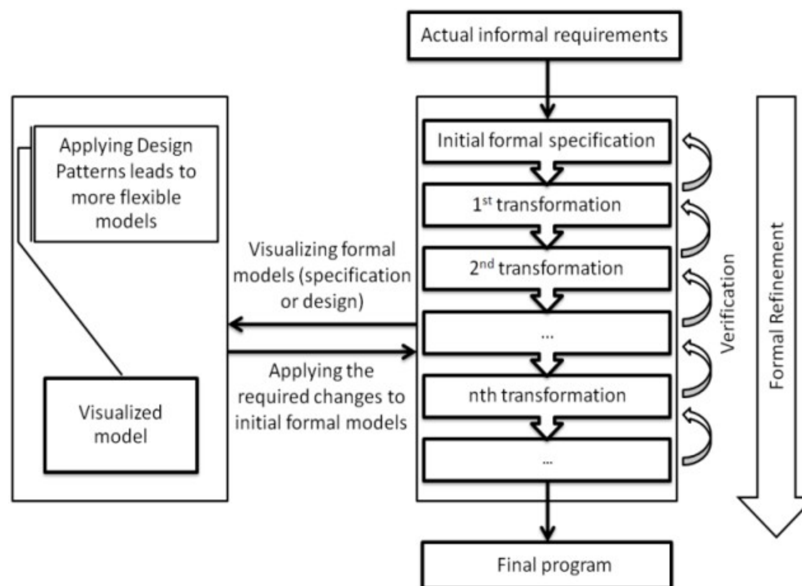

Figure2. A schematic view of the proposed approach

Software includes two aspects: structure (static) and behavior (dynamic) [24], [29]. 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 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.

3 The New Proposed Approach

There has been an evolution in the way of transforming the models [14], [21], [25]. 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 will propose a bidirectional meta-model-based transformation [15] between UML [20] and Object-Z [8]. To do so, the two languages will be defined in terms of their meta-models. 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. Such a transformation preserves 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.

As previously mentioned, we aim to visualize the behavioral aspect of formal models of software to facilitate the possibility of using narrative techniques and principles of software engineering during software development process to produce reliable, yet flexible 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 [11], [30]. There are three types of design patterns, including structural, creational, and behavioral patterns [10], [11]. According to the goal of this work, we focus on the behavioral patterns which shift your focus away from flow of control to let you concentrate just on the way objects are interconnected. For more clarity, we analyze the software behavior from the view points of mediator, observer, and state design patterns.

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.

We use the state design pattern in either of the following cases: 1) an object behavior depends on its state, and its behavior must be changed at run-time depending on its current state, 2) operations have large and multipart conditional statements that depend on the object state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The state pattern puts each branch of the conditional in a separate class. This lets you treat the object state as an object in its own right that can vary independently from other objects. State pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class. Summarily, the required activities to visualize the software behavior form the viewpoint of state pattern include: 1) systematic discovery and elicitation of the objects that their functions are dependent on their states, 2) visualizing the discovered objects as appropriate candidates for review, as well as 3) automatic proposing of suitable times for software behavior review from the viewpoint of the state 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 those UML diagrams that cover software behavior. These diagrams include: 1) state diagram which illustrates the software behavior through identifying the states and the events that result in changing the states [23], [24], 2) activity diagram which visualizes the sequence, transposition, and concurrence of the operations' execution of the various functions at run-time, 3) sequence diagram which represents the interactions among a set of objects and their relationships [23].

An appropriate evaluation method helps determine the overall effects or outcomes of the new approach in relation to promised objectives. This method may indicate whether the promised objectives were met, and 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 system using the proposed method.

Case studies are one of the most common evaluation approaches in software engineering [8], [31]. Based on this approach, the effectiveness and performance of the new methods introduced by researches and studies are empirically measured from the viewpoints of predefined evaluation criteria in one or more case studies. Besides,

in the mentioned case studies, the effectiveness and performance of new proposed approaches are compared with existing related ones (if any). Indeed, another purpose of defining the multi-lift system, addition the goal discussed in section 1, is to provide a suitable test bed to empirically assess the new approach. Currently, the semi-formal and formal specification of the multi-lift system is manually produced and presented in Appendixes A and B. As previously mentioned, proposing the new approach in integrating the UML and Object-Z is left for future work. After that, the new approach will be applied on the formal specification that is manually produced. Then the output will be compared with the existing semi-formal specification. The more these two models are similar, the more the new approach will be precise. So by performing this experiment, we will empirically demonstrate the correctness of the proposed approach as well as the amount of increasing the flexibility of the developed system. Moreover, as mentioned before, we intend to propose a meta-model-based transformation approach. So we will define a formal and systematic transformation between the two languages at the meta-level. In this way, we can prove the correctness, precision, and completeness of the transformation process mathematically. There are also some attempts to measuring software flexibility rather than case studies [32].

4 Conclusion and Future Works

The widespread use of semi-formal methods in mainstream software development leads to highly flexible software. However, they do not take a precise approach to software development in domains where reliability is absolutely crucial, such as safety-critical and high integrity systems. Their semantics are not well defined. Formal methods offer the ability to specify and verify software using mathematical logic. They have precise semantics, allowing for less ambiguous models of systems to be designed. However their use has not been widely adopted due to the mathematical nature of the languages.

This work proposes a new approach to integrate visual and formal models to exploit their advantages in producing reliable, yet flexible software. To do so, this work intends to present a meta-model-based transformation between UML and Object-Z. These 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 can provide a precise, consistent, and complete transformation between a visual model in UML and a formal model in Object-Z. Visualizing the formal models prepares an appropriate ground to revise them from the viewpoints of design patterns which in turn leads to produce high quality software. In the multi-lift case study, we will illustrate how the meta-model-based transformation enables us to create a UML visual representation of an Object-Z specification and vice versa. 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.

Integrated methods research 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. Integrated methods, like formal methods, never really caught-on. A detailed study is given in [1]. The works that have already been done 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], [5], [8], [12], [22]. 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.

There are several attempts to formalize the UML [15], [18], [26], [28], [31], [33]. None of them achieves a full UML formalization. Instead, they focus on a restricted subset of the language, mostly class and state diagrams. Most approaches do not even attempt to formalize all features of a diagram, focusing on those features that are appropriate for the purpose of the approach. Despite some improvements, the integrations that have been proposed are still not practical, which limits their engineering value. Moreover, the integration still falls behind the vision of very specialized software engineering methods. There are also several attempts to integrate design patterns into mainstream software development process [11] as well as to formalize design patterns [4], [9], [15-16]. Generally, despite some improvements in integrating patterns and formalism, the approaches that describe patterns formally are limited in their expressibility and reuse mechanisms.

References

1. Am?lio, N.: Generative frameworks for rigorous model-driven development. PhD thesis, Dept. of Computer Science, University of York (2006)
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS), pp. 436--450 (2007)
3. Bj?rner, D.: Software Engineering 3: Domains, Requirements, and Software Design. Springer (2006)
4. Blazy, S., Gervais, F., Laleau, R.: Reuse of specification patterns with the B method. In: ZB 2003, Turku, Finland, LNCS, vol. 2651, Springer, pp. 40--57 (2003)
5. Bouquet, F., Dadeau, F., Groslambert, J.: Checking JML specifications with B machines. In: ZB 2005, LNCS, vol. 3455, Springer, pp. 434--453 (2005)
6. Charatan, Q., Kans, A.: Formal Software Development: From VDM to Java. Palgrave Macmillan (2004)
7. Charette, R.N.: Why software fails, IEEE Spectrum. vol. 42(9), pp. 42--49 (2005)
8. Duke, R., Rose, G.: Formal Object-Oriented Specification Using Object-Z. MacMillan Press (2000)
9. Flores, A., Moore, R., Reynoso, L.: A formal model of object-oriented design and GoF design patterns. In: FME 2001, LNCS, vol. 2021, pp. 223--241, Springer (2001)
10. Freeman, E., Freeman, E., Kathy Sierra, B.: Head First Design Patterns. O'Reilly Media, First edition (2004)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Pattern: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, Fifth printing (1995)
12. Jackson, D., Schechter, I., and Shlyakhter, I.: Alcoa: the Alloy constraint analyzer. In: International Conference on Software Engineering, Limerick, Ireland, pp. 730–733 (2000)
13. Jackson, D.: Software Abstractions: logic, language and analysis. MIT Press (2006)
14. Kessentini, M., Wimmer, M., Sahraoui, H., Boukadoum, M.: Generating Transformation Rules from Examples for Behavioral Models. In: Second International Workshop on Behavior Modeling: Foundation and Applications, Paris, France (2010)
15. Kim, S., Carrington, D.: A formal meta-modeling approach to a transformation between the UML state machine and Object-Z. In: ICFEM 2002, LNCS, vol. 2495, Springer, pp. 548--560 (2002)
16. Kim, S., Carrington, D.: A rigorous foundation for pattern-based design models. In: ZB 2005, LNCS, vol. 3455, Springer, pp. 242--261 (2005)
17. Liang, H., Song Dong, J., Sun, J., Wong, W.: Software monitoring through formal specification animation. J. of Innovations in Systems and Software Eng., vol. 5(4), pp. 231--241 (2009)
18. Miao, H., Liu, L., Li, L.: Formalizing UML models with Object-Z. In: ICFEM2002, Springer-Verlag, pp. 523--534 (2002)
19. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (NIST), 2002, http://www.nist.gov/

20. OMG, Object Constraint Language (OCL). version 2.0, Object Management Group, http://www.uml.org/ (2006)
21. Poernomo, I.: Proofs-as-model-transformations. LNCS, vol. 5063, pp. 214--228 (2008)
22. Polack, F.: SAZ: SSADM version 4 and Z. In: Software Specification Methods: an overview using a case study. FACIT, Springer, pp. 21--38 (2001)
23. Porres, I.: Modeling and Analyzing Software Behavior in UML. PhD thesis, Department of Computer Science, Abo Akademi University, Finland (2001)
24. Pressman, R.: Software Engineering: A Practitioner's Approach. 7th edition, McGraw Hill (2009)
25. Rahimi, S.K.: Specification of UML Model Transformations. In: Third International Conference on Software Testing, Verification and Validation, pp. 323--326, Paris (2010)
26. Razali, R., Snook, C., Poppleton, M., Garratt, P.: Usability Assessment of a UML-based Formal Modeling Method Using Cognitive Dimensions Framework. Human Technology: An Interdisciplinary J. on Humans in ICT Environments (2008)
27. Schmidt, D.C.: Model-driven engineering. IEEE Computer, 39 (2), pp. 25--31 (2006)
28. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol, vol. 15 (1), pp. 92--122 (2006)
29. Sommerville, I.: Software Engineering. 8th edition, Addison Wesley, June 4 (2006)
30. Stepney, S., Polack, F., Toyn, I.: Patterns to guide practical refactoring: examples targeting promotion in Z. In: ZB 2003, Finland, LNCS, vol. 2651 of, Springer, pp. 20--39 (2003)
31. Williams, J.R.: Automatic Formalization of UML to Z. MSc Thesis, Department of Computer Science, University of York (2009)
32. Eden, A.H., Mens, T.: Measuring Software Flexibility. IEE Software, vol. 153(3), pp. 113--126. London, UK: The Institution of Engineering and Technology (2006)
33. Evans, A., France, R., Lano, K., Rumpe, B.: The UML as a Formal Modeling Notation. In UML'98: Beyond the Notation, Mulhouse, France, vol. 1618, LNCS, Springer, pp. 336-348 (1998)
34. Rasoolzadegan, A., Abdollahzadeh, A.: Empirical Evaluation of Modeling Languages Using Multi-Lift System Case Study. In MSV'11: The 8th annual International Conference on Modeling, Simulation and Visualization Methods, Las Vegas, Nevada, USA (2011)