Symbolic Representation of Models Improves Model Understanding and Tendency to Use Models – A Position Paper

Mira Balaban

Computer Science Department, Ben-Gurion University of the Negev, ISRAEL
mira@cs.bgu.ac.il

I have been teaching Object-Oriented (OO) Modeling [1] for a few years, then I taught an advanced course on Foundations of Software Engineering (SE) [2] for several years, and recently, I lead a workshop on developing a software engineering application [3]. In the Object-Oriented Modeling course we teach UML using a pragmatic view-point, i.e., emphasize how and when to use various features, based on their semantics and properties. In addition we teach a process of project development, using the taught models. In the Foundations of SE course we teach selected chapters including testing, project development methods, design patterns, refactoring, design by contract, and more. The workshop on project development consists of 5-student groups that collectively develop an application construction, guided by a staff member.

Following such an extensive education, my expectations were that students will recognize the value of early modeling, and will get used to analyze and design software with models, as a standard practice. I was rather surprised to find out that once out of the OO Modeling course, most students do not use models unless enforced. Moreover, somehow, most students live under the impression that models are kind of "annoying documents" that accompany software, and can be written as an afterthought, when software is submitted. This is, for example, the way that most students use models in their final, year long project. In extreme cases students use automatic model creation procedures, available in advanced integrated development environments.

This poor situation of actually using models as documentation rather than for analysis, design and development of software, can be explained by multiple factors, some essential to model nature, and others more pragmatic, based on available tools. On the pragmatic reasons we suggest four reasons:

1. Lack of model-level tool support, at the level of modern compilers.
2. Lack of support for model-code coordination all along the software life cycle. Development environments do not keep models consistent with code. Therefore, models become irrelevant very quickly, and programmers find it difficult to preserve model-code agreement.
3. Software development environments do not support advanced modeling features.
4. There are no commercial model processing tools that enable model testing. USE [4,5] is an academic tool for UML/OCL class diagram validation, using instance creation and verification.

On the more essential reasons, we point on two reasons:

1. There is a technical difficulty in creation of visual models, while writing symbolic specifications poses no technical problems. The heavy-use of visual tools does not help either.
2. The third essential reason involves the declarative nature of analytic knowledge and the difficulty of abstraction specification: Declarative expression of (*what is*) knowledge is harder and requires deeper understanding and analytic capabilities, in comparison to procedural (*how to*) knowledge [6]. Model abstraction and constraint specification is way more demanding than straightforward code writing. Indeed, it is not surprising that software testing is more popular than contract specification, as in [7,8].

Model properties: Formal definition and visualization

Traditionally models have been developed as visual notations, since they were intended as intuitive sketches of business level abstractions. Models were understood as a means for intuitive explanations, either to non-professionals, or as a means for initial analysis or design of software. In most cases, there was no intention of being coordinated with applications along their life cycle, and smooth model evolution was not a goal.

Some exceptions are the *Statecharts* model, and *Description logics*. The first, was developed as a visual *Domain Specific Language* for automated realtime systems. It was formally designed, implemented, and used in actual industrial applications [9,10,11,12]. Description logics [13] emerged from the traditional semantic networks [14] and the Frame-based system KL-ONE [15], in AI. They are intended for conceptual automated reasoning and do not have a visual representation. They serve as the basis for the OWL family of semantic web languages [16].

The need for symbolic definition of models

The emergence of the Model Driven Engineering (MDE) approach has changed the picture with respect to models for software analysis and design. Automatic translation, processing and code coordination require well-defined specification of semantics, processing, evolution and management techniques. Models are not anymore just pictures drawn on a wipe board. Indeed, the OMG and other industries supported a wide development of modeling languages, standards, and tools. Nevertheless, while usage of models is growing in modeling Domain Specific Languages, they are still scarcely used in the process of application development.

The thesis I try to raise in this position paper is that if models need to live and evolve with the software, they should be appealing not only to naive users, but also to the community of developers. The pragmatic reasons listed above suggest four points for improving the performance of model processing tools, but this is not within our reach. The two essential reasons involve the technical difficulties posed by visual specifications and the essential difficulty of abstract and declarative specification. While the latter reason requires long range education, the problem of visual specification can be solved relatively

easily. Developers, and even students, with some program writing experience, prefer symbolic code writing over drawing visual models. One class in the Object-Oriented Modeling course that I taught, has used the USE system for validation of UML/OCL class diagrams. I noticed that after mastering the USE symbolic specification of class diagrams, students preferred the symbolic writing, over visual model drawing.

The obvious conclusion is that while visualization is certainly advantageous as a presentation layer, it is not appealing to programmers. That is:

> Every MDE modeling language must have symbolic syntax, in addition to its concrete visual syntax. Teaching of modeling languages must concentrate on the dual existence of the visual and the symbolic syntax.

The expectation is that the design of modeling languages using symbolic syntax will improve their theoretical status. More concretely, the expectations are as follows:

1. The distinction and inter-relationship between concrete and abstract syntax will be defined.
2. It will be clarified that models are not visual presentation layers on top of textual code, but rather abstraction layers on top of more concrete specifications.
3. The use of symbolic syntax will enable employment of standard language techniques for defining semantics and implementing application tools. The Description Logics experience shows that symbolic theoretical language development can yield deep and advanced theoretical and practical results.
4. The use of symbolic syntax will enable the development of testing tools.

Finally, I hope that making models symbolic will clarify that models are formal representation languages, at a higher abstraction level, rather than just visual, intuitive, not executable pictures of problem aspects. I suggest that modeling courses will follow the USE example, and teach a modeling language using symbolic and visual representation.

References

1. Analysis and Design of Software Systems: (2014)
2. Foundations of Software Engineering. `http://www.cs.bgu.ac.il/~fsen141/Main` (2014)
3. Workshop on a Software Engineering Project. `http://www.cs.bgu.ac.il/~wsep142/Main` (2014)
4. Bremen Database Systems Group: A UML-based Specification Environment- Version 3.0. `http://www.db.informatik.uni-bremen.de/projects/USE/` (2012)
5. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling **4** (2005) 386–398
6. Winograd, T., Flores, F.: Understanding computers and cognition - a new foundation for design. Addison-Wesley (1987)

7. Meyer, B.: Applying'design by contract'. Computer **25** (1992) 40–51
8. Mitchell, R., McKim, J., Meyer, B.: Design by contract, by example. Addison Wesley Longman Publishing Co., Inc. (2001)
9. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Transactions on Software Engineering and Methodology (TOSEM) **5** (1996) 293–333
10. Harel, D., Politi, M.: Modeling reactive systems with statecharts: the STATEMATE approach. McGraw-Hill, Inc. (1998)
11. Gery, E., Harel, D., Palachi, E.: Rhapsody: A complete life-cycle model-based development system. In: Integrated Formal Methods, Springer (2002) 1–10
12. Harel, D., Kugler, H.: The rhapsody semantics of statecharts (or, on the executable core of the UML). In: Integration of Software Specification Techniques for Applications in Engineering. Springer (2004) 325–354
13. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation and Applications. (2010)
14. Wood, J.: What's in a link. Readings in Knowledge Representation. Morgan Kaufmann (1985)
15. Brachman, R.J., Schmolze, J.G.: An Overview of the KL-ONE Knowledge Representation System*. Cognitive science **9** (1985) 171–216
16. McGuinness, D.L., Van Harmelen, F., et al.: OWL web ontology language overview. W3C recommendation **10** (2004) 2004