A Pragmatic Approach to Formalizing Object-Oriented Modeling and Development

Xiaoping Jia
Division of Software Engineering
School of Computer Science, Telecommunication, and Information Systems
DePaul University
Chicago, Illinois, U.S.A.
E-mail: `jia@cs.depaul.edu`

Abstract

Despite recent developments in formal methods, formal methods have yet to enter the mainstream of software industry. In this paper, we first examine the pragmatic obstacles of making formal methods practical and usable. Then, we present a pragmatic approach to integrate a popular Object-Oriented modeling notation UML and a popular formal notation Z. Our approach is distinctive in its emphasis on overcoming the practical obstacles. We also present a light-weight supporting tool for our approach. A prototype of the supporting tool is completed. It allows us to demonstrate the feasibility and capability of our approach.

1 Introduction

In recent years, the Object-Oriented software development methodology has become widely accepted by the software industry. Many of its underlying technologies have become matured, including

- Object-Oriented analysis and modeling techniques and notations, such as OMT, and Booch Notation.
- Object-Oriented design patterns; and
- Object-Oriented programming languages, especially C++, and Java;

Many of the promises of Object-Oriented development methodology, such as increased extensibility, reuseability, and maintainability, have been proven true in practice. A sign of the maturity of the field is the recent convergence of Object-Oriented modeling notations in the form of the *Unified Modeling Language* (UML)[1]. As exemplified by UML, the Object-Oriented modeling notations are becoming more and more precise in syntax and semantics, so that Object-Oriented analysis and design models can be built with more rigor and formality. However, even in UML, many aspects of Object-Oriented analysis and design models remain informal or semi-formal, such as the data and operations specifications. Therefore, the gulf between the Object-Oriented analysis and design models and their implementations still remains, so do many problems of the traditional development methods. The main consequences are:

a) Problems in the analysis and design models may not be discovered until the implementations are completed, which will result in high maintenance costs.

b) Only limited automation of the detail design and implementation is possible. In other words, only *skeletal* implementation can be automatically generated from Object-Oriented analysis and design models.

On the other hand, researchers in *formal methods* have long sought to bring rigor and formality into the entire software development process, and to ground the software development process soundly on mathematical foundations. Formal methods promise increased reliability of software systems, and the potential of automating the software development process. Much progress has been made in formal methods in both theory and practice in recent years, most notably, the development of the Z notation [11], its application in industrial-strength projects [4], and the availability of supporting tools such as ZTC [8] and ZANS [7]. It is seemingly promising and tempting to combine the strengths of Object-Orientation and formal methods. However, despite numerous attempts to make formal methods Object-Oriented[10], formal methods have yet to enter the mainstream of the software industry with unfulfilled promises.

In this paper, we first discuss some of the major pragmatic obstacles in making formal methods acceptable and usable in practice. Then, we demonstrate an approach to integrating Object-Orientation with formal methods that intends to overcome these obstacles. Our approach consists of two components: an *augmented* Object-Oriented modeling language (AML), and a supporting tool that is capable of animating the Object-Oriented models in AML and synthesizing fully functional and robust implementation from the models. AML is based on a widely accepted semi-formal Object-Oriented modeling notation, the UML [1], and the widely used formal specification notation, Z[11]. The goal of our approach is to make the adoption formal methods in practice less radical, and more as a complement to the conventional methods rather than a replacement, hence to give formal methods a better chance to enter the mainstream of the software industry. The capability and feasibility of the approach presented here is fully demonstrated by a prototype tool – *Venus*[9].

2 Pragmatic Obstacles

The have been laudable efforts by the formal methods community to demystifying formal methods and to convince practitioners that formal methods are practical [3]. However, as the formal methods community zealously appeals to the practitioners, it fails to fully understand and address the practical concerns of practitioners. Since a reasonably solid foundation has been established for formal methods, it's time to focus more on the pragmatics of formal methods. As a beginning, we identified the following major pragmatic obstacles to the wide-spread use of formal methods.

Notational barrier Many notations used in formal methods have their origins in mathematics. The symbols, conventions, and rules differ, drastically sometimes, from ones that are commonly used in programming languages. This notational difference creates a barrier for software engineers to learn and adapt to formal methods.

Communication barrier Formal notations used in specifications create a communication barrier between software developers and their customers, who usually are not familiar with formal notations and refuse to base contracts on formal specifications.

Radical changes Adoption of formal development processes entails radical changes in development methodologies, techniques, and project management.

Burdens of proof A key benefit of using formal methods is the possibility of guaranteeing correct designs and implementations with respect to formal specifications. However, it requires arduous and tedious efforts to prove a significant number of proof obligations. Most engineers, not just software engineers, are neither accustom to nor trained for proving theorems.

Front-loaded costs In order to realize the benefits of formal methods, it requires significantly more efforts at the early stages of the project life-cycle comparing to the traditional development process. Although the long term cost saving may outweigh the initial cost increase, the significant increase in front-end costs makes it difficult to justify to one's customers and may render one's business uncompetitive.

Inadequate tools Although some supporting tools for formal methods are available, they are vastly inadequate in terms of their capabilities and usability.

Expendable quality The emphasis of formal methods has always been the greatly enhanced quality of software systems, not productivity. However, the reality in the software industry is that productivity and product usability are often more crucial than quality.

There is little possibility for formal methods to overcome all these obstacles. However, we intend to show that it is possible for formal methods to overcome many of these obstacles, and to make formal methods practical and usable.

3 Light-Weight Formal Methods

Traditionally, formal methods promote a rigid software development process that consists of the following:

- using formal notations for specification and design;
- applying theories of design refinement, including data and algorithm refinement;
- discharging proof obligations manually or with the help of theorem provers.

All these efforts lead to the promised correctness of designs and implementations with respect to the formal specifications. It's easy to see such a process has little chance to overcome the pragmatic obstacles and to have wide-spread use. In order to overcome the pragmatic obstacles, it is necessary for formal methods to seek a different role and process.

Recently, a new school of thoughts has emerged — the *light-weight* formal methods approach [5]. Although it is not yet well-defined and fully demonstrated, it represents a significant departure from the traditional formal methods approach. As characterized by Jackson and Wing, the light-weight approach is to accept partialities in notation, modeling, analysis, and composition.

The approach we present here is to partially demonstrate that light-weight formal methods is usable, feasible, and beneficial. The role of formal notations in our approach is not to verify that implementations conforms to their specifications, but only to precisely describe the system behaviors. Our light-weight formal methods approach intends to bring partial automation in the following aspects:

- Extensive mechanized analysis of Object-Oriented analysis/design models for early problem detection.

- Partial animation of the Object-Oriented analysis/design models to demonstrate the system behaviors before the implementation is started.
- Substantial automation of detail design and implementation by synthesizing fully-functional and robust code, that guarantees the absence of many, although not *all*, design and implementation errors, and the conformance of the implementations to their specifications.

In the following sections, we will discus the augmented Object-Oriented modeling notation used in our approach and the architecture and characteristics of a supporting tool.

4 The Augmented Notation

To large extent, the notational barrier was created unnecessarily. Most of the concepts and structures denoted by formal notations have their counterparts in programming languages. The choice of unusual symbols and concrete syntax in formal notations high-lights the dissimilarities between the formal notations and commonly used programming languages, and covers the underlying similarities.

The recent popularity of Java proves that the popularity of a language has a lot more to do with its simplicity and similarities with other commonly used languages than semantic completeness, expressiveness, and mathematical elegance. The most crucial function of a notation is to serve as a means to communicate, both among human beings and between human beings and computers. So the most important goal in designing a notation is to enhance, not obscure, communication.

4.1 Integrating UML and Z

Our approach is not to design a completely new notation, but rather to integrate two existing notations: UML and Z. UML is chosen because of it's popularity among software practitioners, its simple, intuitive, and widely accepted graphical notations, and its relatively precise semantics. An inadequacy of UML is that it leaves data types and operations unspecified or specified in any notations, formal or informal. This provides a perfect opportunity for formal notations to augment UML. The choice of Z is, admittedly, an imperfect one. It is nevertheless the best received formal notations in software industry [2], and it is a relatively simple and manageable formal notation for practitioners.

In order to retain the familiarity and the original graphical appeal of UML, and to achieve synergetic integration of the two notations, we attempt to:

- minimize changes and extensions to the original notations;
- avoid duplication of same information in two different notations, so that the two notations will be complementary and non-interfering;

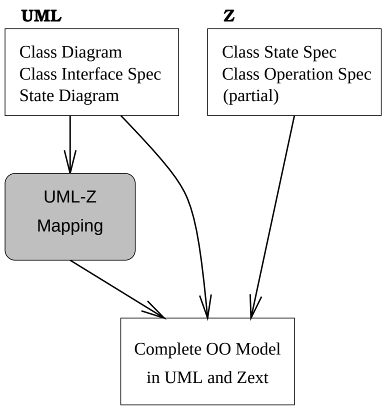

Figure 1: Integration of UML and Z.

- allow analysis and manipulation of the models as a whole not separated parts, so that inconsistencies between components in different notations can be identified.

A subset of UML diagrams forms the foundation of AML, and Z notation is used to compensate the inadequacies of UML. Specifically, AML consists of the following components:

- class diagrams in UML;
- class interface specifications in UML;
- state diagrams in UML;
- class state, or data, specifications in Z; and
- class operation specifications in Z.

The UML diagrams are used to describe the high-level structures of the Object-Oriented models, such inheritance and association relationships among classes, and the class interfaces. The Z notation is used to describe the details of the models that are left unspecified in UML, such as the composition and the data types of class states, and the behavior of class operations.

The graphical notations of UML are adopted without change in AML. While Z abstract syntax is adopted, a number of different concrete syntax is provided to minimize notational differences between Z and commonly used programming languages. The concrete syntax is also customizable to cater to different styles and preferences.

The integration of the two notations is accomplished by mapping the information in UML diagrams to an slightly extended Z notation, called *Zext*. The approach is illustrated in Figure 1.

4.2 Extensions to Z

Since the Object-Oriented concepts and structures are modeled in UML and not duplicated in Z, only a few extensions of Z are required.

4.2.1 Lexical extension

A common way to refer to an operation `op` of class `C` is `C::op`. Although it is possible to use an encoding scheme to refer to the same operation as, say C_op, in Z, it is confusing and may cause name collisions with other unrelated names. Thus, a minor lexical extension is made in Zext to allow *name1*`::`*name2* to be a legal name.

4.2.2 Type system extension

In order to support class types and subtypes, extensions to Z type system is necessary. Instead of introducing a complicated type system similar to ones in many Object-Oriented programming languages, we choose to adopt a simple, restricted, but pragmatically adequate type system.

Zext extends the plain Z in the following way to handle classes:

- The state of a class `C` is specified by a Z state schema of the same name, i.e., `C`.
- Each operation `op` of class `C` is specified by a Z operation schema named `C::op`. Schema `C::op` usually includes either Δ`C` or Ξ`C`.
- The type of class `C` is schema type `C` in Z.
- Inheritance can be handled by schema inclusion, i.e., if class `C1` is a subclass of class `C`, then the state schema of `C1` will *implicitly* include the state schema of `C2`;
- A binary association, including has-a and use-a relations, between classes `C1` and `C2` can be handled by *implicitly* include a relation declaration in the state schemas of both `C1` and `C2`. The multiplicity of the association determines whether the relation declaration is a simple relation, function, or injection, *etc.*
- If class `C1` is a subclass of class `C`, then the type `C1` is compatible with the type of `C`.

Both lexical and type system extensions can be easily incorporated in existing Z tools.

5 The Supporting Tool

5.1 Technical Obstacles

The lack of sophisticated and usable tools is clearly a hindrance to making formal methods practical [2]. There are two well-known major technological barriers in building supporting tools for formal methods:

- Many formal notations, including Z, are non-executable. While executability is extremely desirable, the executable subsets of these formal notations are usually too restrictive. Executing formal notations, such as Z, is intractable in general.
- Using formal methods in software development usually involves a significant amount of theorem proving. Theorem proving in general is also intractable.

Although there is little chance that we can overcome these obstacles, it is still possible to build *effective* and *useful* supporting tools within the boundaries set by these technological barriers. Our approach is to build a *light-weight* supporting tool with the following characteristics:

- It does not intend to solve all the problems, certainly not the most difficult ones, but rather to solve a large majority of common problems that occur in practice. In other words, it intends to automate a significant portion of the formal development process.
- When it fails in some cases, it provides the following remedies:
  - It is extensible and capable of learning new solutions from human engineers.
  - It is capable of integrating formally developed components with manually developed components using conventional approaches.

A prototype of the light-weight supporting tool, *Venus*, has been developed. It intends to demonstrate that the light-weight approach is feasible, and practically useful. It also hope to show that by automating a large majority of formal development tasks, formal development can be as productive as conventional development approaches.

5.2 Overview of Venus

Venus fully supports the Augmented Object-Oriented Modeling language, AML, discussed earlier. It has the following capabilities:

- Partial but extensive analysis of the Object-Oriented models, both the UML diagrams and Z data/operation specifications;
- Partial but extensive animation of the Object-Oriented models to serve as prototypes;
- Design refinement with a fixed but extensive collection of data structures and algorithms from a pre-built library.
- Automatic synthesis of fully functional and robust C++ code;

5.2.1 Analyzer

The analyzer performs the following tasks:

1. Extracting pertinent information from the UML diagrams and translating it into Z segments.
2. Combining the Z segments from Step 1. with the data and operation specification, and sorting them according to definition-before-use order.

3. Analyzing combined and sorted specification in Zext. Analysis performed include:
   - type checking;
   - consistency and completeness analysis.
4. Analyzing UML diagrams for consistency.

The *consistency* analysis checks the satisfiability of the post-conditions of each operation, and the preserving of class invariants. The *completeness* analysis checks whether the pre-conditions are *True*, i.e., all cases are covered, for each user-level operation. The consistency and completeness analysis uses on a *light-weight theorem prover*. It is heuristics-based, and fully automatic. It is not designed to prove the most difficult theorems or all the theorems, rather to prove a significant percentage of theorems that commonly occur in consistency and completeness analysis. The work on the light-weight theorem prover is still in progress. Again, the goal of model analysis is not to guarantee model consistency and completeness, but to identify anomalies in early stages.

5.2.2 Animator

The animator is an extension of ZANS, a animator for plain Z [7, 6]. It animates a large subset of Z specifications. A key characteristics of our animation approach is that the set of executable specifications is not fix but user extensible. The animator turns the Object-Oriented models into prototypes, albeit crude. However, it serve the purpose of demonstrate the system behaviors at an early stage.

5.2.3 Code Synthesizer

Boost of productivity comes from the substantial automation of the design and implementation of systems by the Code Synthesizer, which offers the following capabilities:

- It allows designers to choose from a fixed but extensive collection of design and implementation schemes that are proven sound and efficient. Then the chosen design and implementation schemes will be applied automatically.
- Instead of generating *skeletal* code like most of the currently available CASE tools, *Venus* will generate complete and functional C++ code. The generated code can be compile and run without modifications.
- It guarantees:
  a) conformance of the generated code to the specifications; and
  b) absence of many common errors in the generated code, including memory leaks, dereferencing invalid pointers, object slicing, etc.
- The generated code can be integrated with manually coded C++ components. It also supports iterative development by preserving manually coded components through iterations.

The output of *Venus* Code Synthesizer include

- C++ source code;
- makefiles and build scripts; and
- documentation of the generated code.

5.3 Overcoming pragmatic obstacles

The *Venus* tool attempt to address some of the pragmatic obstacles discussed earlier in the following ways:

Ease the communication barrier The communication barrier is difficult to overcome, but it is possible to compensate that with the capability of animating the formal specifications to demonstrate the specified behaviors, and the capability of more extensive analysis and error detection of specification.

Less radical changes To make the adoption of formal methods less radical, we use formal methods to complement rather than replace conventional development approaches, and use formal notations only when the conventional notations are inadequate. We also support the integration of formal and conventional approaches.

Remove the burden of proof The burden of proof is placed on the automatic theorem prover not on human engineers. When the automatic theorem prover fails, human engineers will takeover using formal or conventional approach. The resulting components can be integrated with tool produced components.

Increase productivity The automation of detail design and implementation will significantly increase the productivity comparing to the same phases in the conventional process.

The core functionalities of *Venus* are completed. An alpha release version is available.

6 Experiments

A number of small to medium scale case studies have been conducted. Early experiments focused on the feasibility of the approach and the capability of the supporting tool, not the usability of the approach and the tool. The case studies use requirements drawn from real life problems including library information systems, inventory management systems, and student registration and recording systems. The size of the case studies range from 200 to 1,500 lines of combined UML and Z specifications, excluding comments. (The UML diagrams are stored in a text format, the lines of the text files containing the diagrams are counted.) The experiments are done by people who are familiar with the notation and tool. These case studies show that

- The AML is natural and sufficient, so far, in capturing the real-life requirements.
- All the specifications can be completely animated. This is to say that for each piece of requirement there is a natural way to specify it so that it can be animated.
- Complete, functional, conforming, and error free C++ code can be generated automatically from all the specifications. The ratio between the lines of specifications and the lines generated C++ code is typically between 1:6 to 1:10.
- For someone who is equally familiar with AML and C++, the development effort required using our approach is less than what required for developing the same system using conventional approaches in C++.

In summary, the early experiments are encouraging. The prototype is still under development. Larger experiments of real-life projects are planned in the near future.

7 Future Work

Our work continues in the development of a more complete, capable, and usable tool in the following aspects:

- seeking more effective strategies and tactics for the light-weight theorem prover;
- using the theorem prover not only in analysis, but also animation, code synthesis, and optimization;
- facilitating customized design refinement in data and algorithms;
- integration with graphical user interface (GUI) builders.

8 Conclusions

In this paper, we presented an approach that intends to overcome many pragmatic obstacles in making formal methods practical. We did not present any technical or theoretical breakthroughs in either formal notations, or development methods, or algorithms. Nor is this our primary concern. What we present is a pragmatic compromise between theory and practice to make an impact on practice within the boundary of current technology. This is what *engineering* supposed to be.

This is an initial, and admittedly still inadequate, effort to address the practical obstacles of making formal methods practical. In this paper, we have demonstrated an approach to integrating Object-Orientation with formal methods. With the support of light-weight tools, we attempt to overcome the pragmatic obstacles of using formal methods, and to make a case for adopting formal methods in practice not solely on increased reliability but on increased productivity as well.

We do not attempt to break the technological barriers of building formal methods supporting tools. However, we intend to show that it is possible to build effective and usable supporting tools for formal methods and to make a significant impact on practice within the boundaries set by the technological barriers.

Acknowledgments

This work is partly sponsored by the National Science Foundations under contract CCR-9410236.

References

[1] G. Booch and J. Rumbaugh. *Unified Methods for Object-Oriented Development, Documentation Set, Version 0.8*, 1995. Available from Rational, Inc.

[2] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial application of formal methods. Technical report, National Institute of Standards and Technology, 1993. NISTGCR 93/626.

[3] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[4] I. Houston and S. King. CICS project: Experiences and results from the use of z in ibm. In *Proc. VDM'91 – Formal Software Development Methods, LNCS No. 552*, pages 588–596, 1991.

[5] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21, Apr. 1996.

[6] X. Jia. An approach to animating Z specifications. In *Proc. 19th Annual Int'l Computer Software and Applications Conf.*, Dallas, Texas, USA, Aug. 1995.

[7] X. Jia. *A Tutorial of ZANS — A Z Animation System*, June 1995. Available via anonymous ftp at `ise.cs.depaul.edu`.

[8] X. Jia. *ZTC: A Type Checker for Z Notation, User's Guide, Version 2.02*, June 1995. Available via anonymous ftp at `ise.cs.depaul.edu`.

[9] X. Jia. *Venus: A C++ Code Generation Tool — A White Paper*, July 1996. Available via anonymous ftp at `ise.cs.depaul.edu`.

[10] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. Prentice Hall, 1994.

[11] J. Spivey. *The Z Notation, A Reference Manual*. Prentice Hall International, second edition, 1992.