The UML as a Formal Modeling Notation

R. France
Department of Computer Science & Engineering
Florida Atlantic University, Boca Raton, Florida, USA

A. Evans
Department of Computing
Bradford University, Bradford, UK

K. Lano
Department of Computing
Imperial College, London, UK

B. Rumpe
Department of Computer Science
Munich University of Technology, Munich, Germany

1 Introduction

The popularity of object-oriented methods (OOMs) such as OMT [16] and the Fusion Method [5], stems primarily from their use of intuitively-appealing modeling constructs, rich structuring mechanisms, and ready availability of expertise in the form of training courses and books. Despite their strengths, the use of OOMs on nontrivial development projects can be problematic. A significant source of problems is the lack of semantics for the modeling notations used by these methods. This can lead to the following problems:

- Understanding of models can be more apparent than real. A stated strength of OO modeling notations is their intuitive appeal, reducing the effort required to read and understand the models. The lack of precise semantics for OO notations can result in situations where a reader's interpretation is not the same (or is not consistent) with the model creator's interpretation. Unless the reader (e.g., a customer) and creator (e.g., a software analyst) explicitly communicate their interpretations there is the danger that both will walk away with inconsistent views of the modeled structure and/or behavior without realizing it. In the cases where the reader is responsible for implementing the behavior specified in the models the result can be an implementation that is not consistent with the view of the models. To compound the problem, it is difficult to rigorously establish consistency of the models and their implementations. Communication also becomes problematic when the concepts used to explain the modeling constructs have no precise meaning, and are themselves subject of misinterpretation.

- Developers can waste considerable time resolving disputes over usage and interpretation of notation. In practice, the use of OOMs can lead to situations where it is not clear to the users of the methods which of a number of opposing views on interpretation and usage is appropriate. Appeal to textbooks is often not helpful because of the lack of meaningful examples and the informal descriptions provided. Over time developers may develop a more precise variant of the notation tailored to their development environment. These more precise interpretations are developed by consensus and are based on interpretations provided by the tools used, the experiences of the users, and the experiences of others they have come in contact with. This process, if it exists within an organization, is very informal, and often the more precise interpretations

[FEL+98] R. France, A. Evans, K. Lano, B. Rumpe.
The UML as a formal modeling notation.
In: Computer Standards & Interfaces. Volume 19, p. 325-334. 1998.
www.se-rwth.de/publications

are not explicitly available to all engineers, rather they exist primarily in the heads of the more experienced developers. Within a single organization, different teams may develop different variants of the modeling. This can result in communication problems when team members move from project to project.

- Rigorous semantic analysis is difficult. In practice OO models are validated and verified informally (in requirements/design reviews). These informal techniques are often inadequate; they cannot be used to rigorously establish that implementations and models are consistent with each other, and rigorously establish that models capturing different views of the system are consistent with each other. Review meetings can be further enhanced if the notations used have a precise semantics. The results of model validations and verifications can be presented in reviews as evidence of the quality of the models. Such formal analysis serves to increase confidence in the creative results of the modeling activity. Rigorous semantic analysis techniques also facilitate the early detection of modeling errors which considerably reduces the cost of error removal.

- Tool support limited to syntactic concerns. First generation tools supporting OO modeling notations mainly focused on editing and drawing issues. As class diagrams (object models) are based on entity-relationship-diagrams, considerable effort is going into generating code, for example, class frames and SQL tables, from them. Due to the lack of precise semantics there is considerably less support for other modeling notations, usually leading to error-prone hand-made translation into code.

The *Unified Modeling Language* (UML) [12] is a set of OO modeling notations that has been standardized by the Object Management Group (OMG). It is difficult to dispute that the UML reflects some of the best modeling experiences and that it incorporates notations that have been proven useful in practice. Yet, the UML does not go far enough in addressing problems related to the lack of precision. The architects of the UML have stated that precision of syntax and semantics is a major goal. In the UML semantics document (version 1.1) [11] the authors claim to provide a "complete semantics" that is expressed in a "precise way" using metamodels and a mixture of natural language and an adaptation of formal techniques that improves "precision while maintaining readability". The meta-models do capture a precise notion of the (abstract) syntax of the UML modeling techniques (this is what meta-models are typically used for), but they do little in the way of answering questions related to the interpretation of non-trivial UML structures. It does not help that the semantic meta-model is expressed in a subset of the notation that one is trying to interpret. The meta-models can serve as precise description of the notation and are therefore useful in implementing editors, and they can be used as a basis to define semantics, but they cannot serve as a precise description of the meaning of UML constructs.

The UML architects justify their limited use of formal techniques by claiming that "the state of the practice in formal specifications does not yet address some of the more difficult language issues that UML introduces". While this may be true to some extent, we believe that much can be gained by using formal techniques to explore the semantics of UML. On the other hand, we do agree that current text-based formal techniques tend to produce models that are difficult to read and interpret, and, as a result, can hinder understanding of UML concepts. This latter problem does not diminish the utility of formal techniques, rather, it obligates one to translate formal expressions of semantics to a form that is digestible by users of the UML notation.

In this paper we discuss how experiences gained by formalizing OO concepts can significantly impact the development of a precise semantics for UML structures. We motivate an approach to formalizing UML concepts in which formal specification techniques (FSTs) are used primarily to gain insights to the semantics of UML notations. The goal of our proposed UML formalization is to

produce a clear, precise expression of the UML notation semantics that can be used by users of the UML notation. In section 2, we give an overview of other works on the formalization of OO modeling concepts and relate it to our attempts at formalizing UML. In section 3 we describe the goals of the PUML project and outline and motivate our formalization approach. In section 4 we illustrate our formalization approach, discussing the structural view (class diagrams). We conclude in section 5 with a summary and a list of some of the open issues that have to be tackled if our approach is to bear meaningful results.

2 Formalizing OO Concepts: An Overview of Integrated Methods

Three general approaches to formalizing OO modeling concepts can be identified: *supplemental*, *OO-extended formal language*, and *methods integration* approaches.

In the supplemental approach parts of the informal models that are expressed in natural language are replaced by more formal statements. The most developed example of the supplemental approach is Syntropy [6]. In Syntropy, OMT-like models are annotated with mathematical expressions. Formal expression of annotations makes models more precise and less ambiguous, but the semantics of graphical constructs are not necessarily precisely defined in these approaches: in Syntropy only the semantics of the static data model is defined. In order to be industrially usable for rigorous OO development, it must be possible to use the formal semantics underlying the diagrams as the basis for tool-supported transformations, such as refinement steps. In principle the approach allows much (but not all) of the complexities of a formal method to be concealed from users.

In the OO-extended formal language approach, an existing formal notation is extended with OO features. Several OO extensions of formal notations have been proposed in the literature (e.g., Z++ [15] and Object-Z [7]). Often the intent of adding OO modeling concepts to formal notations is to enhance the structuring capabilities of the base formal language. In this respect these approaches have indeed resulted in richer formal notations. Furthermore, incorporating OO concepts into formal notations requires that the OO concepts be formalized. The result is a rich body of work on formal notions of object behavior and some aspects of class structures. From a practical perspective, a problem with the notations developed using this approach, at least initially, is the lack of analysis tools. Also, the models created using these notations are not easy to read, understand and modify because of the large semantic gap between real world concepts and their mathematical representations in the formal notations. Considerable effort is needed to map concepts between the real-world and formal domains.

In the methods integration approach informal OO modeling techniques are made more precise and amenable to rigorous analysis by integrating them with a suitable formal specification notation. A number of integrated OO and formal notations have been proposed (e.g., see [8, 2, 13]). Most works focus on the generation of formal specifications from less formal OO models. The act of formalizing an informal model can reveal significant problems that are easily missed in less formal analyses of the models. Furthermore, the formal specifications produced can be rigorously analyzed, providing another opportunity for uncovering problems.

In an integrated method the formal specifications generated from informal models are intended to reflect the formal interpretations associated with the informal models. In fact, the generation of formal specifications from informal models is only possible if there is a mapping from syntactic structures in the informal modeling domain to artifacts in the formally defined semantic domain. This mapping is used to build interpretations of the informal models. Often, this mapping is informally described (e.g., through examples) and rarely justified in papers describing particular integrated methods. The question of whether the generated specifications indeed capture the intended interpretations of the informal models is difficult to answer without a more formal description of the mapping rules. Another

benefit of formalizing the mapping between graphical and formal constructs is that it can uncover problems with the modeling notations. For example, it can help identify ambiguous and inconsistent structures. In addition, a formalized mapping can be used as the basis for defining semantically well-formed informal models.

The existence of a mapping does not necessarily mean that the generation of formal specifications from OO models can be completely automated. The OO models often do not contain all the information needed to generate a formal model, and if they do, the information is not expressed in precise terms (hence, their informality). In most integrated methods the generation of formal specifications from OO models requires the developer to supply the additional information in a suitable formal format. Other approaches automatically generate what they can and it is then up to the developer to complete the formal specification.

A significant barrier to the practical use of integrated methods is the need for users to directly manipulate the generated formal specifications (e.g., to complete and analyze the specifications). Users of integrated methods must have working knowledge of the formal notations. The gap between the intuitive meaning of graphical OO models and their representations in the formal models is wide. As mentioned previously, this means that considerable effort is needed to comprehend the formal models. Realizing the full benefits of integrated formal methods requires users of the methods to have in-depth knowledge of both the informal OO and formal specification techniques. Such a user is currently rare in industry.

When a complete semantic mapping from a graphical OO notation into formal semantic domains is possible then the OO notation must itself be treated as a formal notation. This is the basis of the PUML work on formalizing UML. The intent is not to generate formal specifications from UML models, rather, the objective is to develop a more formal version of UML that can be used to build precise and analyzable models. This paves the way for the development of tools that support semantic analysis of OO models, and does not require developers to have knowledge of another formal notation.

3 Towards a Precise UML

The methods integration approaches to formalizing graphical OO modeling techniques outlined in the previous section focus on how traditional text-based formal specification notations can be used in the context of informal, graphical OO modeling techniques. Unfortunately, system analysts/developers in industry find it difficult to express their business rules, information structures, software requirements and designs in existing formal notations. This is because much effort is required to transform real-world concepts to the abstract mathematical structures used by most formal notations. For this reason, understanding formal specifications can also be difficult. One of the strengths of the OO techniques in UML is that they provide constructs that allow developers to build models that have a clear connection to real-world concepts.

Rather than generate formal specifications from informal OO models and require that developers manipulate these formal representations, a more workable approach is to provide formal semantics for graphical modeling constructs and develop rigorous analysis tools that allow developers to directly manipulate the OO models they have created. Formal specification techniques can be used to explore and gain insights into appropriate formal semantics for graphical modeling constructs. Once identified and formally stated, the semantics can be re-expressed in a form that is digestible by users of the OO modeling notation (e.g., natural language, mixture of text and graphical representations).

The UML provides a unique opportunity for us in this respect. It is still very much a notation that can benefit from formalization of its constructs. We are currently embarking on a collaborative project, called the *Precise UML* (PUML) project, that has as its focus the formalization of core UML concepts. A major objective of the project is to develop a formal *reference manual* for the UML.

This will give a precise description of core components of the language and provide inference rules for analyzing their properties. In developing the reference manual we will build upon the semantics given in the UML semantics document [11] by using formal techniques to explore the described semantic base. Such formalization is very likely to uncover problems related to inconsistent, incomplete, and ambiguous descriptions of the UML semantics. The insights we gain through formalization will be used to develop precise descriptions of the semantics that will be presented in the reference manual in a readable form. In this section we motivate and give the objectives of the *Precise UML* (PUML) project.

3.1 PUML objectives

Given its role as a standard modeling notation, it is imperative that the UML have a well-defined, fully explored semantics. A formal semantics would make the UML a formal modeling notation and paves the way for its use in rigorous OO system development. Exploring the semantic base of UML with formal techniques can be beneficial for the following reasons:

- Formalization allows one to derive and explore consequences of particular interpretations. Such exploration can yield insights that can help determine the appropriateness of interpretations. Problems related to incomplete, inconsistent, and ambiguous interpretations can be unmasked through rigorous analysis. Similarly, formal characterizations of semantic domains and semantic mappings can be analyzed to uncover problems and yield insights.

- Variants of the semantics can be obtained by relaxing and/or tightening constraints on semantic models. This paves the way for the development of a variety of semantic models for UML constructs that can be used in various modeling contexts.

- Overall, the formalization of UML constructs can lead to a deeper understanding of OO concepts, which, in turn, can lead to the development of more sophisticated semantic analysis tools, and to the more mature use of OO technologies.

A primary object of the PUML project is to develop a formal characterization of core UML concepts. The intent is that the semantics we provide for the core concepts will provide a base for developing semantic variations of the UML (e.g., semantics tailored to system and product modeling contexts). As pointed out above, such variants can be obtained by modifying the characterizations of the semantic domains and mappings. For example, there are many forms of aggregation, most of which are applicable only to specific modeling contexts. In the UML two forms of aggregation are distinguished: the strong form called *composite* aggregation and the weak form simply called *shared* aggregation. In our formalization of the core concepts we will formalize these two forms; semantic variations of aggregation can be obtained by modifying these formal characterizations.

Some of the concepts identified as core in the UML include: types, values, operations, behaviours, associations, hierarchy and inheritance. However, other concepts such as collaborations, refinement and design patterns will also be considered in due course. At present, a draft denotational semantics of some of the above concepts has been developed [4] using the Z notation [18]. Already, the task of producing this specification has identified some interesting and subtle ambiguities in the UML meta-model.

The appropriateness of identified semantics for UML constructs cannot be determined by formal analysis only. It is necessary to gain feedback from expert OO modelers and industrial users of the notations. The current UML documents reflect a significant amount of OO modeling expertise. Formalization builds upon the expertise as discussed above. The results of formalization can also benefit from expert feedback. Such feedback is necessary at least to determine the significance of the

insights gained and the appropriateness of the semantic mappings. In order to achieve the PUML objectives it is necessary to collaborate with industrial users of the UML and expert OO modelers. The PUML group will utilize mechanisms that facilitate expert feedback on the formalization results (e.g., publishing results and soliciting feedback on the PUML web site (*http://www.comp.brad.ac.uk/*) and on relevant listservers).

The semantics obtained through formalization will be described in a precise and readable form in a reference manual. The reference manual will also provide descriptions of useful semantic variants of constructs. A particular objective of the project is to ensure that the reference manual is accessible to mainstream software engineers. As mentioned above, we hope to achieve this by re-expressing the formal semantics in terms of a suitably expressive language. This could be achieved, for example, by using a mixture of notations such as an enhanced version of the UML meta-model, the Object Constraint Language (OCL) [10], and precise natural language statements (precise in the sense that they are readable re-expressions of more formally represented concepts).

3.2 Roadmap to Formalization

In this subsection, we give an overview of the formalization approach and discuss some of the issues that have to be tackled.

In the introduction we discussed *why* a formalization of UML description techniques is useful. From that discussion we can derive the following two requirements for a formalization:

1. A formalization must be complete, and as abstract (but meaningful) and understandable as is possible.

2. The formalization of a heterogeneous set of description techniques has to be integrated to allow the definition of dependencies between them.

This does not mean that every syntactical statement must have a formal meaning. Annotations or descriptions in prose are always necessary for documentation, although they do not have a formal translation.

A clear and precise notion of what constitutes a system and how it should be denoted in OO terms is needed. Such a characterization provides a firm base for subsequent formalization activity. For this reason, the first step in our formalization approach is to formalize the notion of a system in terms of its constituent parts, interactions, and static and behavioral properties. This activity requires the formalization of modeling concepts that are independent of specific modeling notations and techniques. In the OO modeling realm this is possible because objects have certain properties that are independent from the modeling techniques, and are thus intrinsic to "being an object". Examples of such properties include having attributes or having sequentially invocable methods. In [14] and [17] a system model is defined, and used, as described in papers such as [3] and [17], as a basis for formalizing OO diagrams. Such a system model can also be viewed as the semantic domain, as it is used as the domain of the semantics mapping.

The second step is to formally define the abstract syntax of the graphical OO notations. For that purpose, the UML meta-model [9] is well suited, as it gives a precise notion of the abstract syntax. The UML semantics document also provides well-formedness rules expressed in the expression language OCL. These rules are used to determine whether a UML structure can be associated with a well-defined meaning. This part of the UML semantics is already well defined and we do not anticipate that our formalization will add much significant insights to this aspect of the semantics. However, the meta-model primarily aims at readability for the user. For a formalisation, an even more abstract characterisation of the syntax, e.g. using math or Z (as we did below) is a better starting point.

The third step of the formalization is concerned with defining the mapping between the syntactic domains (as characterized in step 2) and the semantic domains. The mappings relate syntactic constructs, such as class names, to semantic ones, like actual classes. The system model formally defines the set of all possible systems. A document of a given description technique is then defined by relating its syntactic elements to elements of a system, such as the existing set of classes, or other structural or behavioral entities. The semantics of a document is then given by a subset of the system model. This subset of the system model consists exactly of all systems that are correct implementations of the document.

It is an important advantage, to separate formalization of the semantic domain and the semantics mapping, by defining the system model explicitly, because this leads to a better understanding of the developed systems, allows the user to understand what a system is independently of the used notation, and allows to add and integrate new OO diagram forms.

If the semantic domain is carefully defined, then the mapping between syntax and semantics becomes easier. If there is a larger coincidence between the syntactic and the semantic domain, then the mapping may even be the identity to some extent. For example, if the notion of class does exist in the semantics, either because it was defined (e.g. a schema using Z), or it already exists in the used language (e.g. Object-Z, Z++), identity can be used in this case. Thus, syntactic and semantic domain may overlap to some extent. For convenience, we will allow such overlapping, but is important to be aware of this situation to avoid confusion. It is also important that syntactic and semantic concepts have the same meaning. For example, if the syntax allows the denotation of classes with publicly accessible attributes, then a mapping to classes without such attributes is not appropriate.

It clearly is beyond the scope of this paper to carry out these steps. In the next section we give a few examples of the formalizations that can be produced in our approach.

4 A Formalization Example

In this section, we formalize the meaning of a classifier in terms of the set of *object instances* that it describes. To keep the example small, we show only the part of a formalisation that deals with classes and attributes. Please note, that this section demonstrates the formalisation approach, especially the separation of syntactic and semantic domains. It is however not intended to be already a considerable contribution to the formalisation of UML.

4.1 Abstract Syntax

The first step in formalizing the meaning of a classifier is to describe its abstract syntax.

First, it is assumed that there are the given sets:

$$[ClassifierName, AttributeName]$$

from which the set of all classifer and attributes names can be drawn.

A classifier has a name, and a set of attributes:

$$\begin{array}{|l} \textit{Classifier} \\ \hline name : ClassifierName \\ attributes : \mathbb{F}\, AttributeName \\ \hline \end{array}$$

At any point in time, a UML model will contain a set of uniquely named classifiers:

$$
\begin{array}{|l}
\textit{AbstractSyntax} \\
\hline
classifiers : \mathbb{F}\, Classifier \\
\hline
\forall c_1, c_2 : classifiers \mid c_1 \neq c_2 \bullet c_1.name \neq c_2.name \\
\hline
\end{array}
$$

The constraint of the schema states that each classifier must have a unique name.

4.2 System Model

In order to give meaning to classifiers, values must be assigned. In the UML, a classifier is viewed as defining a set of possible object instances. This is the 'system model' that we adopt for our formalization.

The given types $ObjectName$ and $AttributeLink$ describes the set of all object identities and values of interest:

$$[AttributeLink, ObjectName]$$

An object is owned by a classifier, has a unique identity, and maps a set of attributes to their values:

$$
\begin{array}{|l}
\textit{Object} \\
\hline
owner : ClassifierName \\
name : ObjectName \\
attributes : AttributeName \pfun AttributeLink \\
\hline
\end{array}
$$

At any point in time, the meaning of a UML model is a finite set of unique object instances:

$$
\begin{array}{|l}
\textit{SystemModel} \\
\hline
objects : \mathbb{F}\, Object \\
\hline
\forall o_1, o_2 : objects \mid o_1 \neq o_2 \bullet o_1.name \neq o_2.name \\
\hline
\end{array}
$$

4.3 Meaning function

This section describes the meaning of classifiers as a mapping from classifiers to object instances:

It is assumed that there is a relationship between attributes and their values:

$$
\begin{array}{|l}
value\_of : AttributeLink \pfun AttributeName \\
\end{array}
$$

The following function describes the meaning of a classifier. It maps a classifier a set of possible combinations of object instances ($Objects$), whose attribute values conform to that permitted by

the classifier. By 'conform' it is meant that the values of each object's attributes conform to those permitted by the attribute of the owning classifier:

$$
\begin{array}{|l}
\mathcal{M}_{classifier} : Classifier \pfun \mathbb{P} Object \\
\hline
\forall c : Classifier \bullet \\
\quad \mathcal{M}_{classifier}(c) \subseteq \{o : Object \mid \\
\quad\quad o.owner = c.name \land \\
\quad\quad o.attributes = \{a : c.attributes;\ v : AttributeLink \mid \\
\quad\quad\quad\quad value\_of(v) = a\}\}
\end{array}
$$

Similarly, a function to describe meaning of a collection of classifiers (the abstract syntax) can be defined using the above function:

$$
\begin{array}{|l}
\mathcal{M}_{syntax} : AbstractSyntax \pfun \mathbb{P} SystemModel \\
\hline
\forall a : AbstractSyntax \bullet \\
\quad \mathcal{M}_{syntax}(a) = \{s : SystemModel \mid \\
\quad\quad s.objects = \bigcup \{c : a.classifiers \bullet \mathcal{M}_{classifier}(c)\}\}
\end{array}
$$

It maps an *AbstractSyntax* to the set of possible *SystemModel*'s that may be derived from the meaning of the abstract model.

4.4 Advantages of Formal Notations

The advantage of using a formal notation like Z to formalize components of the UML is that all the usual facilities for Z are available for:

- Type checking the components
- Proving properties about the components
- Giving precise rules for manipulating the components

In addition, by permitting incremental specification, it is possible to generate the meaning of parts of the UML seperately and then combine them to produce a semantics of the whole model. In this way, a compositional model of the UML can be built, which has important consequences for proof and refinement of UML models.

The main disadvantage of using Z however, is that faithfully mapping the UML Semantics to Z can result in very verbose specifications. Intuitively, expressing the UML semantics using a meta-model is much easier for a non-formalist to understand and comprehend than a Z specification. As an example, consider the UML meta-model description of the model presented above. The class diagram representing the syntax and partial semantics of the model is depicted in figure 1 and the OCL expression required to ensure that the values of an instance conform to those permitted by the attributes of its owning classifer or class is:

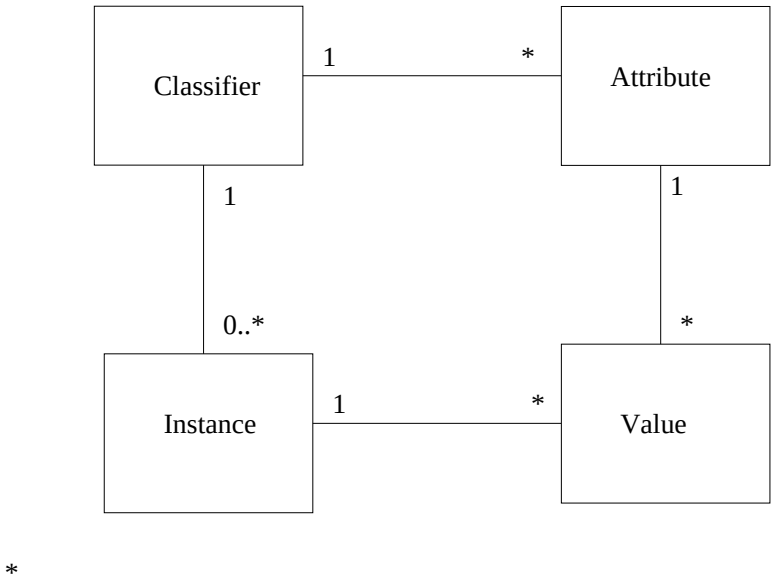

Figure 1: Class diagram representing syntax and partial semantics

```
Classifier
self.instances − > forall(i :  Instance |
  (i.values − > forall(v :  Value |
     v.attributes = self.attributes)) and
  i.values− >sum = self.attributes->sum)
```

The constraint states: (1) that each value of an instance must belong to the set of values permitted by an attribute belonging to the owning classifier; (2) the number of values an instance may have must be equal to the number of attributes of the owning classifier.

However, although the UML description is simpler (and shorter) than the Z version, it suffers from any means of proving properties about the components. This is a critical shortcoming which will need to be addressed if a precise semantics and proof system for UML is ever to be developed using the meta-model approach.

5 Summary and Open Issues

In this paper we have presented our motivation for formalizing the UML. The objective of our efforts is to make the UML itself a precise modeling notation so that it can be used as the basis for a rigorous software development method. The benefits can be summarized as follows:

- Lead to a deeper understanding of OO concepts, which in turn can lead to more mature use of technologies.
- The UML models become amenable to rigorous analysis. For example, rigorous consistency checks within and across models can be supported.
- Rigorous refinement techniques can be developed.

An interesting avenue to explore is the impact a formalized UML can have on OO design patterns and on the development of rigorous domain-specific software development notations. Domain-specific

UML patterns can be used to bring UML notations closer to a user's real-world constructs. Such patterns can ease the task of creating, reading, and analyzing models of software requirements and designs.

An integrated approach to formalisation of UML models is needed in order to provide a practical means of analysing these models. Current work on compositional semantics [1] has used techniques for theory composition to combine semantic interpretations of different parts of an OO model set.

Some of the other issues that have to be addressed in our work follows:

- How does one gauge the appropriateness of an interpretation of UML constructs? In practice an 'accepted' interpretation is obtained by consensus within a group of experts. Formal interpretations can facilitate such a process by providing clear, precise statements of meaning.
- Should a single formal notation be used to express the semantics for all the models? The advantage of a single notation is that it provides a base for checking consistency across models, and for refinement of the models. This is necessary if analysis and refinement is done at the level of the formal notation. On the other hand, if the role of the formal notation is to explore the semantic possibilities for the notations, and analysis and refinement are carried out at the UML level, then there seems to be no need to use a single formal notation.
- Are the identified core concepts sufficient?
- How can the formal semantics of the UML be best presented to non-formalists?

It is anticipated that as our work progresses additional issues that will have to be tackled will surface.

To solve these problems the authors have started a project that deals with the formalisation of core elements of UML.

References

[1] J. Bicarregui, K. Lano, and T. Maibaum. Objects, associations and subsystems: A heirarchical approach to encapsulation. In *Proceedings of ECOOP 97, LNCS 1489*. Springer-Verlag, 1997.

[2] Robert H. Bourdeau and Betty H.C. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.

[3] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the unified modeling language. In Satoshi Matsuoka Mehmet Aksit, editor, *ECOOP'97 Proceedings*. Springer Verlag, LNCS 1241, 1997.

[4] T. Clark and A. Evans. Foundations of the Unified Modeling Language. In *Proceedings of The Second Northern Formal Methods Workshop*. Springer-Verlag, 1997.

[5] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, Englewood Cliffs, NJ, Object-Oriented Series edition, 1994.

[6] Steve Cook and John Daniels. Let's get formal. *Journal of Object-Oriented Programming (JOOP)*, pages 22–24 and 64–66, July–August 1994.

[7] Roger Duke, Paul King, Gordon A. Rose, and Graeme Smith. The Object-Z specification language. In Timothy D. Korson, Vijay K. Vaishnavi, and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice Hall, 1991.

[8] Robert B. France, Jean-Michel Bruel, and Maria M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modeling Environment. *To appear in the Journal of Object-Oriented Programming (JOOP)*, 1997.

[9] The UML Group. UML Metamodel. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, September 1997.

[10] The UML Group. UML Object Constraint Language Specification. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997.

[11] The UML Group. UML Semantics. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997.

[12] The UML Group. Unified Modeling Language. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997.

[13] J. Anthony Hall. Using Z as a specification calculus for object-oriented systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 290–318. VDM-Europe, Springer-Verlag, New York, 1990.

[14] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model - . In Jean-Bernard Stefani Elie Naijm, editor, *FMOODS'96 Formal Methods for Open Object-based Distributed Systems*, pages 323–338. ENST France Telecom, 1996.

[15] Kevin C. Lano. Z^{++}, an object-orientated extension to Z. In John E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 151–172. Springer-Verlag, 1991.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[17] Bernhard Rumpe. *Formal Method for Design of Distributed Object-oriented Systems.* Ph.d. thesis (in german), Technische Universität München, 1996.

[18] J. Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, Englewood Cliffs, NJ, Second edition, 1992.