Defining Formal Semantics for the Unified Modeling Language

William Andreopoulos
Department of Computer Science
University of Toronto
February 2000

Abstract

The Unified Modeling Language (UML) has become a standard diagrammatic notation for describing object-oriented (OO) models and designs. UML suffers from a lack of a formal (or precise) semantics. UML semantics is mostly described using lengthy paragraphs of often ambiguous informal English, or is totally missing. This paper examines the research conducted at present by the PUML (Precise UML) group towards formalising UML. We present the three main approaches that have been identified for UML formalisation; the supplemental approach, the OO-extended formal language approach, and the methods integration approach. We illustrate the approaches with specific examples from published sources. Finally, we explain the reasons for which UML formalisation is such a difficult subject and we discuss what the future might look like as a result.

Contents

Chapter 1

Introduction

The Unified Modeling Language (UML) is rapidly becoming a global industry standard diagrammatic notation for expressing and describing object-oriented models and designs. UML is a set of description techniques for specifying, visualising and documenting object-oriented (OO) systems. UML is based on intuitive and easy to understand diagrammatic notation, whose aim is to be understandable to system developers and customers.

The latest UML Version 1.1, supports the use of a semi-formal constraint language, called the Object Constraint Language (OCL). The OCL was proposed as a standard expression language for expressing additional constraints on UML models; such constraints cannot be conveniently expressed on UML models using UML's diagrammatic form alone.

UML and OCL are semi-formal notations. This means that UML and OCL have a rather precise syntax but an informal semantics; although much of the syntax of these languages has been defined, semantics is mostly described using lengthy paragraphs of often ambiguous informal English, or is missing entirely. Thus, the interpretation of the meaning of UML diagrams and its structure is imprecise. A discussion of the concepts of syntax and semantics is given in Appendix A.

UML and OCL suffer from a lack of a formal (or precise) semantics. A formal semantics for a language gives the precise meaning of each construct of the language in question, using a mathematically-based formal notation, such as Z or Object-Z or Larch. Many researchers, motivated by their wish to develop practical formal methods, have devoted considerable effort into attempting to define a formal semantics for the UML/OCL. Such attempts can be justified by considering the benefits associated with a formal semantics for UML/OCL:

Interpretation: If at any point there is confusion over the exact meaning of a particular UML component, reference can be made to the formal semantics to verify its interpretation. Thus, if a system is being implemented by someone other than the creator of the UML design, he/she won't implement an incorrect interpretation of the design.

Rigorous Analysis and Design: If concepts related to the model have been expressed using formal description techniques, it is possible to carry out a rigorous semantic analysis of a diagram, so as to rigorously evaluate and systematically implement a system. In practice, OO diagrams are validated and verified informally. These informal techniques are inadequate, as they cannot be used to rigorously establish that implementations and models are consistent with each other.

Proof: A formal semantics would provide a strong basis for formally analysing and reasoning about UML models. It would allow justified proofs and checks of important properties of a system described in UML, such as safety and liveness properties. Such proofs could be aided by theorem provers and proof checkers. Automatic proof techniques could also be developed this way.

Extendibility: A formal semantics would enable the soundness of any extensions to the UML to be verified.

Equivalence and Consistency: It would provide an unambiguous basis from which to compare and contrast UML with other techniques and notations, and for ensuring consistency between its different components.

OCL Clarification: A formal semantics for OCL would provide the means for clarifying the meaning of the diagrammatic and textual notation, and also to refine and improve this notation.

The current UML is not sufficiently formal to realize many of these benefits. Little consideration has been paid to important issues such as proof and rigorous tool support. This paper examines the research conducted at present towards formalising UML/OCL, by describing the progress that has been made in this area, and the problems that have been encountered. Section 2 presents an overview of this work, by describing the different approaches taken by researchers, in their attempts to formalise specific subsets of UML/OCL. Section 3 concludes with a comparison of the different approaches, an explanation of the reasons for which UML formalisation is such a difficult topic, and a discussion of what the future might look like as a result of a formalised UML/OCL.

Chapter 2

UML Formalisation

2.1 Overview of Work

Different approaches have been provided for formalising some aspects of UML diagrams and OCL. The Precise UML (PUML) group is the primary research group concerned with formalising UML. The PUML was formed in 1997 with the aim to develop a formal semantics for a core subset of UML. It is also concerned with the development of tools that support rigorous analysis and refinement of UML models.

The PUML group has identified three general approaches for the formalisation of the Object-Oriented (OO) modelling concepts of UML [1]:

The Supplemental Approach: In this approach, the informal UML model is expressed directly using a formal specification language (e.g. LSL).

The OO-extended Formal Language Approach: In this approach, an existing formal notation (e.g. Z) is extended with OO features (e.g. Object-Z), in an attempt to formalise the UML metamodel.

The Methods Integration Approach: In this approach, informal OO modelling techniques are made more precise and amenable to rigorous analysis by integrating them with a suitable formal specification notation (e.g. Z).

Each of these approaches can in turn be subdivided into more specific approaches. Figure 2.1 gives a hierarchical tree illustrating all the possible paths that can be followed for formalising a UML model.

It is important to realise that each approach has focused on the formalisation of specific subsets of the UML. Almost all approaches have attempted to formalise the class diagram element of the UML notation. However, different approaches have also attempted to formalise other elements of the UML notation, such as sequence diagrams and the OCL. Next we describe each of

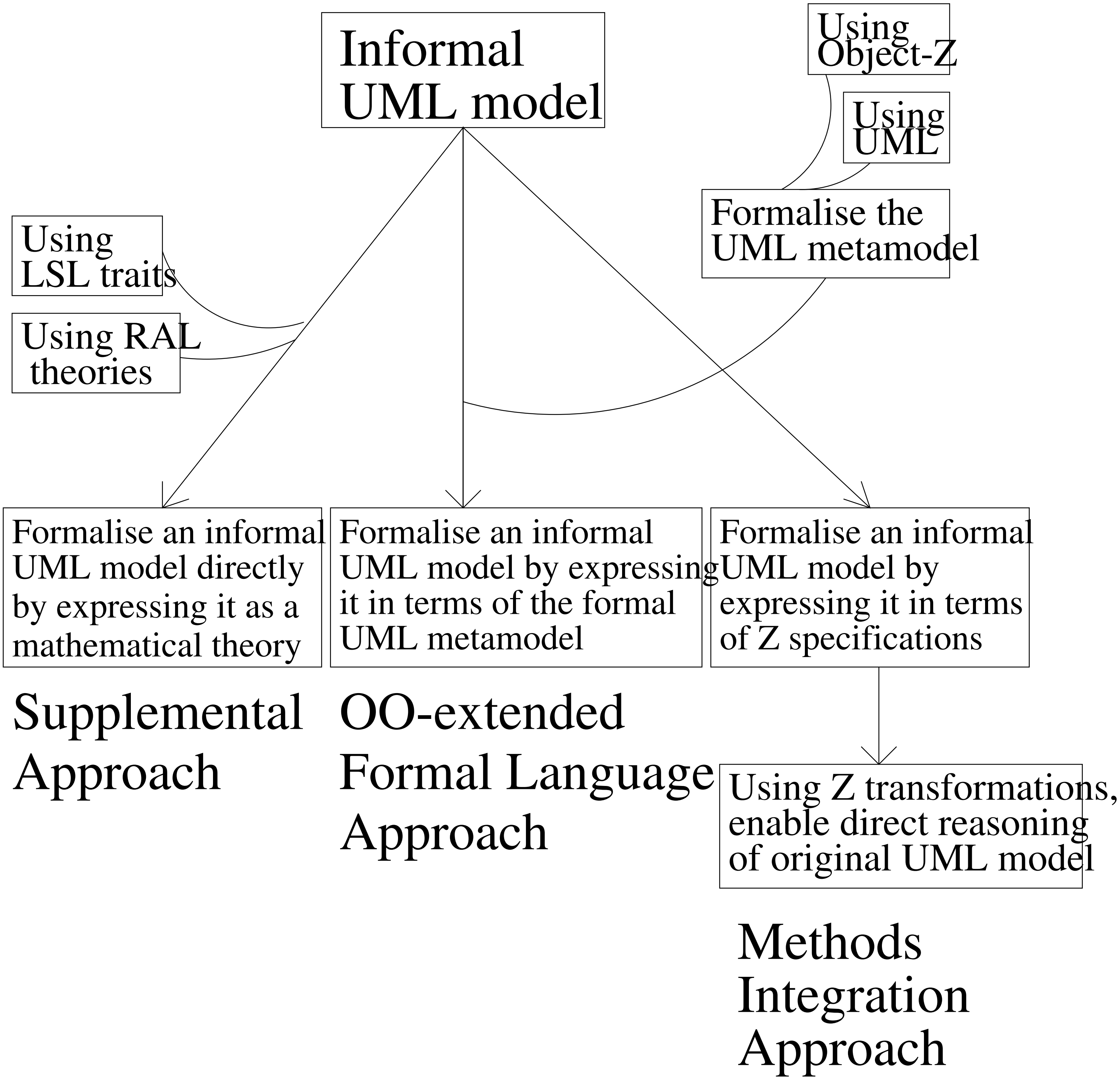

Figure 2.1: Possible approaches for formalising a UML model

these three approaches in detail. We describe the formalisation of OCL last, because OCL is a textual notation as opposed to the graphical notation of UML.

2.2 The Supplemental Approach

The supplemental approach attempts to directly express the semantics of a UML model in a formal language such as Z or LSL. The view taken is that a UML model is a visual representation of a mathematical theory. Such a theory provides vocabulary for precisely capturing the semantics of a UML model. Thus a formal semantic representation of a UML model can be given in terms of theories in a suitable logic.

Theories can be used to represent classes, instances, associations, and general submodels of a UML model. Therefore, these theories describe the features and properties which should be supported by any implementation that satisfies the UML model.

2.2.1 Formalisation Using the Larch Shared Language (LSL)

One approach to expressing these mathematical theories is to use the Larch Shared Language (LSL). LSL has a simple syntax and semantics, with support for theory composition, and it comes with a toolset including a sophisticated proof assistant. With some tools that translate UML models into mathematical theories expressed in LSL, it is also possible to reason about UML models using the Larch theorem prover. Thus, this approach could be used for checking the integrity of UML models. The notation used for defining theories in LSL is called a trait.

LSL has been used to formalise some aspects of UML models, namely class diagrams and OCL expressions. In this section we only examine the formalisation of UML class diagrams. The formalisation of OCL is described in Section 2.5. This work was done by the Odyssey Research Associates [2], and by members of the BIRO project at the University of Brighton, UK [3, 4].

A UML class diagram describes the types of objects in a system and the various associations that exist among them. Class diagrams also show the attributes and operations of each object in a system. It has not yet being attempted to formalise all the semantics of class diagrams, as some details of their semantics have not yet been defined and some are intentionally underspecified. However, enough have been formalised to suggest that the style of model presented can be extended to the full semantics (or, at least, to a useful subset of it adequate for interesting applications).

UML class diagrams can be regarded as visual representations of logical theories. When using this approach, a class diagram is interpreted as a mathematical theory expressed as a single LSL trait. We refer to this LSL trait, which is defined for each class diagram, as 'ClassTheory'.

The LSL trait ClassTheory is built by composing the theories interpreting all the notational elements (object types, associations, actions specifications) derived from the UML class diagram. ClassTheory includes the traits for all the object types, associations, and action specifications in the class diagram. Thus, an important aspect of this approach is to treat every component of a UML class diagram down to the level of individual diagram elements, as distinct semantic entities. This leads to a highly modular approach, allowing reuse of generic semantic entities, resulting in an elegant, transparent semantics for individual models.

For each notational element included in ClassTheory, a separate LSL trait is defined. The declarations and axioms of each such trait, capture the semantics of that specific element. The 'implies' section of each trait contains formal statements of all the properties we want to check. Thus, it has been attempted not merely to capture the semantics of class diagrams, but to make the model rich enough to conveniently express interesting properties of the diagrams.

Therefore, ClassTheory is composed of the following parts:

- The LSL traits (sets of declarations and sets of axioms) associated with each notational element (object types, associations, actions specifications) in the class diagram.
- An 'implies' section determining which properties to check about the class diagram.
- An optional trait declaring a general background theory of objects, associations, etc.
- An optional library of traits that model OCL and its type constructors.

A detailed example illustrating this approach is given in Appendix B.1.

2.2.2 Formalisation Using Real-Time Action Logic

As with the previous approach, a mathematical semantic representation of UML models is given in terms of theories in a suitable logic. Thus, theories can be used to represent classes, instances, associations, and submodels of a UML model. These theories describe the features and properties which should be supported by any implementation that satisfies the UML model.

This approach focuses on formalising UML elements such as class diagrams, statecharts, sequence diagrams, collaboration diagrams, aggregation and qualification. Only formalisation of class diagrams is described here, since formalisation of the other UML elements is similar. This work was mostly done by K.Lano, J.Bicarregui, and A.Evans at the Imperial College and University of York, UK [5, 6, 7].

In this approach, Real-time Action Logic (RAL) is used to create the theories for reasoning about real-time specifications. A RAL theory is similar to an LSL trait. A RAL theory has the form:

theory - Name.

types - local type symbols.

attributes - time-varying data, representing instance or class variables.

actions - actions which may affect the data, such as operations, statechart transitions and methods.

axioms - logical properties and constraints between the theory elements.

Either Z or OCL notation can be used for the axioms in RAL theories, representing the semantics or constraints of UML models.

When formalising a UML class diagram, the whole class diagram can be represented as a RAL theory of the form given in Figure 2.2. Both the classes and the associations between classes must be described using such a RAL theory. Next we explain this theory in more detail.

The 'types' section defines the names of all classes in the class diagram. The 'attributes' section defines the attributes of each class. For each class C, there is also an attribute $\overline{C} : F(C)$ denoting the set of existing instances of C (F is standard mathematical notation for "set of finite sets of"). The 'actions' section defines the actions of each class.

Each association lr of the class diagram is represented in the 'attributes' section, as an attribute $\overline{lr}$ (although this is not shown in Figure 2.2). The representation $\overline{lr}$ signifies the set of pairs in the association lr. For each association in the class diagram, actions *add_link* and *delete_link* are defined in the 'actions' section, to add and remove pairs from this set.

The 'axioms' section defines other properties of the UML class diagram. The cardinality of each association end is a property that could be defined in the 'axioms' section.

A detailed example illustrating this approach is given in Appendix B.2.

theory $\Gamma_{\mathbf{C}}$

types $\mathbf{C}$

attributes $\overline{\mathbf{C}} : \mathbb{F}(\mathbf{C})$

$\quad \mathbf{self} : \mathbf{C} \rightarrow \mathbf{C}$

$\quad \mathbf{att_1} : \mathbf{C} \rightarrow \mathbf{T_1}$

$\quad \ldots$

actions $\mathbf{create_C}(\mathbf{c} : \mathbf{C}) \quad \{\overline{\mathbf{C}}\}$

$\quad \mathbf{kill_C}(\mathbf{c} : \mathbf{C}) \quad \{\overline{\mathbf{C}}\}$

$\quad \mathbf{op_1}(\mathbf{c} : \mathbf{C}, \mathbf{x} : \mathbf{X_1}) : \mathbf{Y_1}$

$\quad \ldots$

axioms

$$\begin{aligned}
&\forall \mathbf{c} : \mathbf{C} \cdot \\
&\qquad \mathbf{self}(\mathbf{c}) = \mathbf{c} \;\wedge \\
&\qquad [\mathbf{create_C}(\mathbf{c})](\mathbf{c} \in \overline{\mathbf{C}}) \;\wedge \\
&\qquad [\mathbf{kill_C}(\mathbf{c})](\mathbf{c} \notin \overline{\mathbf{C}})
\end{aligned}$$

Figure 2.2: Theory of Class C (taken from [7]).

2.3 The OO-extended Formal Language Approach

Unlike the previous approach which attempted to directly express the semantics of a UML model, the OO-extended formal language approach attempts to formalise the UML metamodel instead. The name of this approach comes from the fact that it attempts to extend formal notations (e.g. Z) with OO features (e.g. Object-Z), in order to make them more compatible with the OO concepts of the UML metamodel. In order to describe this approach more precisely, it is first necessary to explain the concept of a metamodel.

2.3.1 Metamodels

The architecture of the UML is based on a four-layer structure, which consists of the following layers: user objects, model, metamodel, and meta-metamodel. Each layer is an instance of its upper layer, i.e. the metamodel layer is an instance of the meta-metamodel layer. The functions of these layers are summarized in the following table.

Layer	Description	Example
meta-metamodel	Defines the language for specifying the metamodel.	MetaClass, MetaAttribute, MetaOperation
metamodel	An instance of the meta-metamodel. Defines the language for specifying the model.	Class, Attribute, Operation, Component
model	An instance of the metamodel. Defines the language for specifying the user objects.	StockShare, askPrice, sellLimitOrder, StockQuoteServer
user objects	An instance of the model. Defines the specific user data.	$< Acme\_SW\_Share\_98789 >$, 654.56, $sell\_limit\_order$, $< Stock\_Quote\_Svr\_32123 >$

Table 2.1: Four Layer Metamodeling Architecture

For this work, we are mostly concerned with the UML metamodel layer. The UML metamodel defines a language for the information that can be expressed during software modeling. In other words, the metamodel is a model of the UML software modeling language. The UML metamodel consists of metaentities, such as Classes, Attributes, Operations, and Components.

A goal of the UML metamodel is to express all aspects relevant to a metaentity (e.g. Classes). These aspects have been identified to be:

Abstract Syntax: an abstract description of the UML metaentities. This is described using UML class diagrams. [8] (In this way the UML bootstraps itself in a manner similar to how a compiler is used to compile itself.)

Static Semantics: well-formedness conditions between the syntactic entities, such as the absence of circular inheritance. This is described using OCL. [8]

Dynamic Semantics: the behavior of the entities of the specification, such as I/O, reaction to stimuli, effect of executing an operation, etc. [8] This is described using mostly natural language (English).

Thus, the UML metamodel describes each of these aspects in a semi-formal manner.

2.3.2 Formalisation Using the Formal Metalanguage Object-Z

This approach has been tested by members of the PUML group, working at the Fraunhofer Institute for Software Engineering and Systems Engineering in Berlin, Germany [8, 9, 10].

This approach claims that since we use a metamodel for the definition of modeling languages such as UML, it is necessary for the metamodel to have a formal semantics as well. A formal metalanguage for the definition of the UML metamodel should support the following OO concepts:

- Objects: encapsulation of an object's internal state. An object defines an entity that might be only accessed in a controlled way.
- Object Identity: the notion of a persistent identity for an object.
- Object Behavior: the ability of objects to evolve in a pre-defined way. Method definitions are used to define object behavior.
- Classes: the ability to describe the common aspects of objects and encapsulate them in a class structure. Objects are instances of a type corresponding to the class. The features of an object are accessible via methods.
- Associations: the ability to declare a structural connection between classes so that a class can access services of the associated class.
- Inheritance: defining a class as an extension of one or more existing classes.

- Polymorphism: the ability to define operations that act upon several distinct classes.
- Constraints: the ability to describe constraints between objects or pre- and post-conditions of objects operations in some kind of predicate logic.

There exist several formal languages that fulfill the above requirements of the metalanguage. Object-Z has been chosen by various researchers as the formal metalanguage, because it supports the OO features listed above and is the most mature object-oriented extension of Z. Object-Z provides the class construct, which is a named box with optional generic parameters. Other components of this construct are a visibility list, inherited classes, type and constant definitions, a state schema, an initial state schema, and operation schemas. Moreover, Object-Z provides history invariant, that is a temporal logic formula to represent more easily the dynamic aspects of the model. A complete description of Object-Z can be found in [11, 12].

As explained above, the UML metamodel consists of metaentities (e.g. Classes). The UML metamodel describes the abstract syntax, static semantics and dynamic semantics of each metaentity semi-formally. The goal of the Object-Z approach is to express all these aspects formally.

When using the Object-Z approach, it is a simple matter to give a formal description for each of these aspects of a metaentity. Abstract syntax and static semantics are defined by Object-Z classes and their state schemas, attributes, and associations. (Constraints on the attributes and associations are described by predicates.) Dynamic semantics is defined by the operations of the Object-Z classes. A comparison of the Object-Z approach to metamodeling to the UML approach to metamodeling is given in Figure 2.3.

	abstract syntax	static semantics	dynamic semantics
UML approach	class diagrams	OCL	natural language
Object-Z approach	Object-Z class specs & class diagrams	Object-Z predicates in class specs	Object-Z operation specs

Figure 2.3: Comparison of UML and Object-Z Approach (taken from [8]).

After the UML metamodel has been formalised by defining it in terms of Object-Z, a UML model can be formalised by describing it in terms of the formal UML metamodel. A short example illustrating the OO-extended formal language approach is given in Appendix C.

2.3.3 Formalisation Using the Meta-modelling Approach

As with the previous approach, this approach also attempts to formalise the UML metamodel. However, this approach attempts to express the formal semantics using UML itself.

The meta-modelling approach to defining the UML semantics and its feasibility was only proposed at a workshop held at ECOOP'99. Although this approach has not been fully implemented yet, it has many advantages which justify discussing it.

The main disadvantage of all the approaches presented until now, is that if one wants to deal with the formal semantics of UML, one also has to deal with the formal language in which the semantics is described (e.g. LSL or Object-Z). Thus in order to understand the semantics of UML, one has to learn a second formal specification language as well. The meta-modelling approach overcomes this problem, by attempting to express the formal semantics using UML. Thus designers do not have to learn another language in order to understand the formal semantics of UML.

A disadvantage associated with this approach is that the expressive power of UML may be unsuitable to express its entire semantics. At this point it is certain that a large part of UML can be expressed using UML alone, however it is possible that not all UML features can be treated successfully.

We give a basic outline of the proposed steps to be taken for this approach. A more detailed explanation of this outline can be found in [13].

1. Develop the syntax of the core meta-modelling language, which is a diagrammatic modelling language for defining other languages. Later, the UML semantics will be defined in terms of this language. This language will definitely include class diagrams and OCL. This core meta-modelling language should be kept as small as possible.

2. Develop the semantics of the core meta-modelling language. This semantics can be given using set theory, temporal and dynamic logic, algebraic specification, streams, process theory, etc.

3. Develop a meta-model description of the semantics of this core meta-modelling language.

4. Transform the syntax of the complete UML language into the UML meta-model. This step converts each syntactic element of UML into an element of the UML meta-model.

5. Finally, relate the meta-model description of the UML syntax, developed in Step 4, to the meta-model description of the semantics of the core meta-modelling language, developed in Step 3.

2.4 The Methods Integration Approach

The previous approaches require that developers generate formal specifications from informal UML models, and then manipulate these formal specifications directly. The methods integration approach attempts not only to provide formal semantics for UML, but also to develop rigorous analysis tools that allow developers to directly analyze the informal OO models they have created, without reference to the underlying formal specifications.

The development of this approach was motivated by the need to be able to prove desired properties of UML models, without having to deal with complicated mathematical notation. This approach has the following goals:

- Develop tools that automatically generate the formal specification of an informal UML model.
- Allow the developer to prove properties of a UML model by manipulating the graphical OO model directly, without any reference to the underlying formal specification language.

This approach is comparable to natural languages. For example Chinese can be understood by using a Chinese-English dictionary. Having a Chinese-English dictionary at hand, one can begin to learn Chinese words, and how to build Chinese sentences. However, one does not want to use a dictionary forever. One's ultimate goal is to be able to directly deal with Chinese, without any translation to English anymore. Similarly, the ultimate goal of this approach is to be able to manipulate a graphical OO model and prove properties about it, without having to deal directly with the underlying formal specification notation.

2.4.1 Roadmap of Formalisation Strategy

The following steps are taken to implement this approach:

1. Identify the core modelling elements of the existing UML semantics to be formalised.
2. For each core modelling element, verify that it has a precise syntax.
3. Define the syntax of the graphical UML notation. The syntax of the UML notation is referred to as language L, and we verified that it has a precise syntax in step 2. [14] The UML meta-model works well as a description of the syntax of UML diagrams, just like grammars are well suited for text. (See Appendix A for a discussion of the concept of syntax).

4. Define the semantic domain. The notion of a system is characterized in terms of its constituent parts, interactions, and static and behavioral properties. This characterization defines the elements of the semantic domain, which we denote by S. [14] The semantic domain can be described using Z. (See Appendix A for a discussion of the semantic domain).

5. Define the semantic mapping function between the syntax L and the semantic domain S. The formal semantics of an informal UML model can be obtained by applying this mapping function to the model [14] (See Appendix A for a discussion of the semantic mapping function).

6. Finally, develop analysis tools for the formalised UML notation. The semantic mapping function defined in the previous step must be implanted in these tools, to enable developers to automatically generate formal specifications for their UML models. Developers can use such tools to constructively enhance, refine and compose UML models, and to verify their properties, without any reference to the underlying formalisations.

2.4.2 Formalisation Using FuZed

A first attempt made in this direction, called FuZed, integrated an informal OO modelling technique, Fusion, with the formal specification notation Z. FuZed was developed by the members of the PUML group A.Evans, R.France, K.Lano, B.Rumpe, J.Bruel, and T.Clark [15, 16, 17, 18, 19].

FuZed permits developers to rigorously analyze their OO models. Analysis of an OO model using the FuZed technique involves developing informal OO Fusion models, and then transforming them to Z specifications. The rules for transforming OO models to Z specifications are implanted in FuZed.

The first modeling phase, involves using the Fusion OO modeling technique to produce UML models. The second step involves analysing the UML models produced during the modeling phase. To analyse UML models means to construct proofs, to determine whether a property being asserted about the model is true or false. For example, given two models of a system, one that represents its specification, and another that represents its design, it must be possible to prove the property that the design is correct with respect to the specification.

In this approach, a proof requires applying a sequence of diagrammatical transformation rules to the UML model diagrams. At each step, using a transformation rule, the previous diagram is transformed into the diagram of the next step of the proof. This continues until the resulting diagram shows that the required property has been proved.

Whenever a transformation rule is applied to a diagram, it must be shown that the resulting diagram is a valid deduction of the original diagram. For this to be shown, every UML diagram needs to have a corresponding Z formal specification. Each diagrammatical transformation rule corresponds to the derivation of a new Z formula, as a result of applying an inference rule to the previous Z formula. Therefore, each sequence of diagrammatical transformation rules corresponds to a sequence of Z specifications, derived by applying inference rules. This continues until the derived Z specification shows that the property being asserted about the original UML diagram is true. A full description of this diagrammatic proof technique can be found in [20].

As explained in the roadmap, it is necessary to define the semantic domain of UML. The next few sections are optional reading, as they present the definition of the UML semantic domain using Z. An example of using this approach can be found in Appendix D.

The Z Formal Specification Language

Z is a traditional formal specification technique, which consists of a notation that directly utilizes mathematical concepts, and provides mechanisms for formally analyzing models expressed in the notation.

Z is used because it is a stable, mathematically precise and structured notation that can express OO modeling concepts in a straightforward manner, and because it is supported by computer-based analysis tools. Z is supported by typecheckers, animators, and theorem proving/checking environments.

Since Z is a well defined and formal specification technique, this approach has the advantage of enabling a formally precise and unambiguous model of the semantics to be written. However, this often happens at the cost of ease-of-use and understandability.

Core Object Model

In this section we identify and formalise some core components of the UML, by using the Z formal specification language. The intention of this formalisation attempt is to describe a subset of UML, as opposed to every aspect of UML. The rest of this section presents a Z specification for some of the key components of a Core UML Model.

The Object Management Group (OMG) has developed a specification for a Core Object Model, with which UML is required to be compliant. This specification is essentially a description of the syntax of a simple object model. The Core UML Model which will be formalised is essentially an extension of this Core Object Model. Thus, the Z specification presented in this section is an extension of the Core Object Model specification.

The Core Object Model specification developed by OMG imposes some requirements on compliant systems, such as UML. Compliant systems must satisfy the following conditions:

- a number of types must be available to the system.
- operations must be associated with each type.
- the types must form a hierarchy.
- inheritance of operations from supertypes must be possible.

Based on the informal semantics of UML, the following additional requirements must be satisfied by any formalisation attempt:

- each type must be associated with a set of possible values. In the case of object types, these values will represent the object instances of the type.
- types must be related by associations.
- operations must be associated with behaviours, and also permit message passing.

Each of the above requirements will be considered next in an attempt to formalise a Core UML Model. This Core UML Model includes the following concepts, that correspond to the requirements presented above:

CoreUMLModel
Types
Instances
Values
Operations
Behaviours
Associations
Hierarchy
Inheritance

The result will be a Z schema for this Core UML Model. (A Z schema is similar to a record. It gives a schema name, and elements which are part of the schema. Schemas also allow to state axioms that must hold between their elements.) The Z schema for this Core UML Model will specifically include several other schemas, each of which will correspond to one of the concepts listed above.

Thus, this approach describes the precise semantics by giving a Z denotation to each of the concepts included in the Core UML Model. This will facilitate the formalisation of important concepts such as behaviour, which is modelled as a set of state changes (transitions). Next we present the Z schemas that have been given to some of the Core UML Model concepts.

Types Z Schema

The Types Z Schema specifies the types that are available.

The types available in a model consist of object types, non-object types (e.g. numbers) and abstract types. To express this, three finite sets of type TypeName are declared in the Types Z schema (TypeName is a set containing the names of all possible types). This schema is shown in Figure 2.4.

$$
\begin{array}{|l}
\textit{Types} \\
\hline
\textit{oTypes}, \textit{nTypes}, \textit{abstract} : \mathbb{F}\,\textit{TypeName} \\
\hline
\mathsf{disjoint}\ \langle \textit{oTypes}, \textit{nTypes} \rangle \\
\textit{abstract} \subseteq \textit{oTypes} \\
\hline
\end{array}
$$

Figure 2.4: Types Z schema (taken from [18]).

The two constraints at the bottom of the schema state that:

- an object type cannot have the same name as a non-object type.
- any abstract type must be an object type.

Instances Z Schema

The Instances Z schema shows that an instance is an instantiation of a type, and has a unique identity. This schema is shown in Figure 2.5.

The constraints at the bottom of the schema state that each instance must be associated with exactly one type.

Hierarchy Z Schema

The Hierarchy Z schema expresses the supertype relationship between object Types. This schema is shown in Figure 2.6.

The constraints at the bottom of the schema state that the collection of object types must form a directed acyclic graph with root Object.

Inheritance Z Schema

The Inheritance Z Schema shows that a type inherits all the operations supported by its supertypes. This schema is shown in Figure 2.7.

$$
\begin{array}{|l}
\textit{Instances} \\
\hline
oTypes : \mathbb{F}\, TypeName \\
oInstances : \mathbb{F}\, Instance \\
instances : TypeName \mathrel{+\!\!\!+\!\!\!\!\rightarrow} \mathbb{F}\, Instance \\
\hline
\mathrm{dom}\, instances = oTypes \\
\bigcup(\mathbf{ran}\, instances) = oInstances \\
\forall\, i, j : \mathrm{dom}\, instances \mid i \neq j \bullet \\
\qquad\qquad instances(i) \cap instances(j) = \emptyset \\
\hline
\end{array}
$$

Figure 2.5: Instances Z schema (taken from [18]).

$$
\begin{array}{|l}
\textit{Hierarchy} \\
\hline
oTypes : \mathbb{F}\, TypeName \\
oSuperTypeOf : TypeName \leftrightarrow TypeName \\
\hline
oSuperTypeOf \in (oTypes \leftrightarrow oTypes) \\
oSuperTypeOf^{*} (\!|\ \{Object\}\ |\!) = oTypes \\
\mathsf{disjoint}\ \langle oSuperTypeOf^{+}, oSuperTypeOf^{+\sim} \rangle \\
\hline
\end{array}
$$

Figure 2.6: Hierarchy Z schema (taken from [18]).

$$
\begin{array}{|l}
\textit{Inheritance} \\
\hline
oTypes : \mathbb{F}\, TypeName \\
oSuperTypeOf : TypeName \leftrightarrow TypeName \\
interface : TypeName \mathrel{+\!\!\!+\!\!\!\!\rightarrow} \mathbb{F}\, Signature \\
\hline
\forall\, S, T : oTypes \bullet \\
\qquad S \mapsto T \in oSuperTypeOf \wedge \\
\qquad (interface(S)) \subseteq (interface(T)) \\
\hline
\end{array}
$$

Figure 2.7: Inheritance Z schema (taken from [18]).

2.5 Formalisation of the Object Constraint Language (OCL)

OCL is a precise textual language that complements the graphical UML language in modelling OO systems. The primary purpose of the Object Constraint Language (OCL) is to express constraints on a UML model, that cannot be expressed using diagrammatic notation. Some types of constraints are pre/post conditions and invariants.

The OCL itself is mostly defined in a semi-formal way by using English text descriptions and by many examples illustrating the intended meaning of OCL expressions. However, due to the semi-formal nature of the OCL definition, the results of some OCL expressions are ambiguous. Thus, a formal and precise semantics for OCL is essential for checking the consistency of the language. This formalisation attempt was done by members of the BIRO project at the University of Brighton, UK [4, 3].

It is possible to formalise some features of OCL using the Larch Specification Language (LSL). The predefined data types in OCL, such as integers, strings, booleans, enumerated types can be easily interpreted in LSL.

The OCL type Boolean can be specified as the Larch sort of boolean values Boolean. The type Integer can be specified as the Larch sort of mathematical integers Integer. The type String can be specified as the Larch sort of character strings String. The basic operations on booleans, integers and strings are predefined in Larch.

In OCL multi-valued expressions are described by collection types. The collection types of OCL are sets, bags and sequences. Sets, bags, and sequences can be constructed in OCL by providing a type parameter for their element type. For example, the type Sequence(Integer) denotes sequence values with integer elements. The type Set(T) can be specified in LSL as Set[T], the Larch sort of sets with elements from T. The type Bag(T) can be specified as Bag[T], the Larch sort of bags with elements from T. The type Sequence(T) can be specified as Seq[T], the Larch sort of sequences with elements from T. The standard operations on sets, bags and sequences are predefined in Larch.

Chapter 3

Conclusion

3.1 Difficulties with Formalising UML

The task of UML formalisation is not trivial and poses many problems. The problems associated with UML formalisation are much larger than those encountered in formalising ordinary computer languages. [21] The main UML features that make formalisation difficult are:

Heterogeneous - UML is a "group of heterogeneous semi-formal notations". [22] This means that UML is not a single language but a series of related languages describing a variety of diagrams for various purposes. The various kinds of UML diagrams use very different notations, such as a variant of entity relationships, state charts, message sequence charts, a kind of Petri nets (activity diagrams), et cetera.

Multiview - The UML model of a system consists of many different diagrams, each one describing a view either of the system or of some of its parts. [22] It may happen that the functionality and some constraints on an operation of a class are in the class diagram, its local behaviour is in a state diagram, its interactions with the objects of another class are in a collaboration diagram, and those with the objects of a third different class are in a sequence diagram.

Extendable - UML offers some ways to the users to extend the kinds of the elements used in the diagrams, as stereotypes, tagged values and constraints. [22] Moreover, the language used for the constraints is not fixed; OCL has been defined as a constraint language, but its use is not mandatory and can be replaced by other languages, including even natural language.

Notation - An important feature of UML is that it is only a notation (or modelling language) and not a method. [22] This implies that UML does not prescribe any particular development process. Thus, UML can be used in different ways by different methods, and a method may consider only parts of UML or different parts in different phases of the development process. Different methods could use the same diagrams intended with different meanings. For example, a method may enforce that all the executions of the objects of a class should be those described by sequence diagrams, another one may just say that such executions are possible, and a third one may use some sequence diagrams to describe prohibited executions.

3.2 Success of each Approach

Of all the approaches presented, the methods integration approach has met the greatest success. The reason is that this approach does not require a developer to be capable to understand a second formal specification language, such as Z. All the other approaches require that developers generate formal specifications from informal UML models, and then manipulate these formal specifications directly. However, the methods integration approach provides rigorous analysis tools that allow developers to directly manipulate their UML models, without reference to the underlying formal notation.

The meta-modelling approach still remains in the experimental stage, and has not been fully tested yet. It does have a potential for success, because it does not require a developer to learn a second formal specification language such as Z. Designers who are not experts in formal specification languages are given the chance to reason about UML without the burden of learning another language. One disadvantage of this approach is that the expressive power of UML may not be suitable for expressing its entire semantics. Another disadvantage is that confusion may arise, since there is only one single language. One always has to point out clearly whether one is speaking of a UML element which is being translated, or a UML element which occurs as the result of a translation.

Formalisation attempts using Larch, the RAL theory and Object-Z have had some limited success in the past, resulting in a rich body of formal systems. However, these formal systems are still too different from current industrial methods to be applied widely in the industry.

3.3 The Future of Software Development with a Formalised UML

A precise UML could benefit software development tools and processes in the future. If UML becomes well-defined, the main added value for the next generation of tools would be to assist the end users to produce consistent and expressive models, including tools for automated consistency checking.

On the other hand, long experience shows that normal software developers will not, in the foreseeable future, be willing to use abstract formal languages and notations to design software, regardless of how theoretically desirable it might be to do so. The main reason that even simple semi-formal languages and tools such as OCL meet great resistance from the typical developer, is that the mental interpretation of logical statements requires a skill in abstract thinking. The only way to get developers to be precise, is to provide tools that require little skill in discrete mathematics and that are very simple and obvious to use.

Nowadays many software developers think they have a precise intuition about all the elements of UML. However, the chances are slim that all these intuitions coincide. Instead, these intuitions are influenced by each developer's education and experience using the UML notation. As tools become more and more mature without the formal UML semantics having become standardized, it will become increasingly difficult to find a standardization that developers can agree upon. This could be a drawback of UML in the future, especially if a new OO modeling notation emerges, with more precisely defined semantics. Thus, defining formal semantics for UML is a very important subject for the future of software engineering, and research is still required in this area.

Bibliography

[1] A. Evans, J.-M. Bruel, R. France, K. Lano, and B. Rumpe, "Making UML Precise", in *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, edited by L. Andrade, A. Moreira, A. Deshpande, and S. Kent, 1998.

[2] M. Bickford and D. Guaspari, Odyssey Research Associates Report No. 10, 1998 (unpublished).

[3] A. Hamie, "A Formal Semantics for Checking and Analysing UML Models", in *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, edited by L. Andrade, A. Moreira, A. Deshpande, and S. Kent, 1998.

[4] A. Hamie, J. Howse, and S. Kent, "Interpreting the Object Constraint Language", in *Proceedings of Asia Pacific Conference in Software Engineering*, IEEE Press, 1998.

[5] K. Lano, "Defining Semantics for Rigorous Development in UML", in *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, edited by L. Andrade, A. Moreira, A. Deshpande, and S. Kent, 1998.

[6] K. C. Lano and A. S. Evans, "Rigorous Development in UML", in *Fundamental Approaches to Software Engineering, FASE'99, Part of Joint European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam*, LNCS, Springer-Verlag, 1999.

[7] K. Lano and J. Bicarregui, "Formalising the UML in Structured Temporal Theories", in *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, edited by H. Kilov and B. Rumpe, pp. 105–121, Technische Universität München, TUM-I9813, 1998.

[8] R. Geisler, "Precise UML Semantics Through Formal Metamodeling", in *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why?*

How?, edited by L. Andrade, A. Moreira, A. Deshpande, and S. Kent, 1998.

[9] R. Geisler, M. Klar, and C. Pons, Technische Universität Berlin Report No. 98-5, 1998 (unpublished).

[10] S. Mann and M. Klar, "A Metamodel for Object-Oriented Statecharts", in *Proc. 2nd Workshop on Rigorous Object-Oriented Methods, ROOM 2*, University of Bradford, 1998.

[11] R. Duke, P. King, G. Rose, and G. Smith, Software Verification Research Centre; The University of Queensland Report No. 91-1, 1991 (unpublished).

[12] D. Carrington *et al.*, "Object-Z: An object-oriented extension to Z", in *Formal Description Techniques (FORTE '89), Vancouver*, pp. 281–296, North-Holland Publishing Co., 1989.

[13] M. Gogolla and M. Richters, "Equivalence Rules for UML Class Diagrams", in *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, edited by J. Bézivin and P.-A. Muller, , LNCS Vol. 1618, pp. 87–96, Springer, 1998.

[14] R. France, A. Evans, K. Lano, and B. Rumpe, Computer Standards & Interfaces **19**, 325 (1998).

[15] J.-M. Bruel and R. B. France, "Transforming UML Models to Formal Specifications", in *The Unified Modelling Language. <<UML>>'98: Beyond the Notation*, edited by P.-A. Muller and J. Bézivin, , LNCS No. 1618, Springer-Verlag, 1998.

[16] R. France, A. Evans, and K. Lano, "The UML as a Formal Modeling Notation", in *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, edited by H. Kilov, B. Rumpe, and I. Simmonds, pp. 75–81, Technische Universität München, TUM-I9737, 1997.

[17] A. Evans and S. Kent, "Core Meta-Modelling Semantics of UML: The pUML Approach", in *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, edited by R. France and B. Rumpe, , LNCS Vol. 1723, Springer, 1999.

[18] A. Evans and T. Clark, "Foundations of the Unified Modeling Language", in *Proc. of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, 23-24 September 1997*, 1997.

[19] A. Evans, R. France, K. Lano, and B. Rumpe, "Developing the UML as a Formal Modelling Notation", in *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, edited by J. Bézivin and P.-A. Muller, , LNCS Vol. 1618, pp. 297–307, Springer, 1998.

[20] A. Evans, "Reasoning with the Unified Modeling Language", in *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)*, 1998.

[21] P. van Emde Boas, "Formalizing UML; Misson Impossible?", in *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, edited by L. Andrade, A. Moreira, A. Deshpande, and S. Kent, 1998.

[22] K. Baclawski, "UML Formalization: A Position Paper", in *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, edited by L. Andrade, A. Moreira, A. Deshpande, and S. Kent, 1998.

[23] B. Rumpe, "A Note on Semantics (with an Emphasis on UML)", in *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, edited by H. Kilov and B. Rumpe, pp. 177–197, Technische Universität München, TUM-I9813, 1998.

Appendix A

Concept of Syntax and Semantics

This appendix clarifies the concept of syntax and semantics and their relationships. Since we are examining and comparing the results of different researchers on the subject of UML formalisation, it is necessary to give a general framework of how to define the syntax and semantics of a notation. [23]

The reason is that although there exists a large amount of excellent theoretical work that gives subsets of the UML a precise semantics, this work often makes some implicit assumptions, which influence the way a formalisation is defined and the results of such a formalisation. Developers tend to use the word "semantics" when they talk about the behavior of a system they develop. But mathematicians and logicians use the word "semantics" when referring to the "meaning" of a notation, regardless of whether this notation deals with structure or behavior. Thus, it is necessary to define our view of "semantics" as well.

A.1 Syntax

This section clarifies the concept of "syntax" and how syntax is represented. The term "syntax" is used whenever we refer to some notation. Syntactical issues focus on a notation (in our case UML), disregarding any intention behind the notation. Specifically, the syntax defines how to use all the different elements of a notation correctly. The syntax is referred to as the language L.

In conventional textual notations, syntax is described by the set of characters used (alphabet) and their possible sequences (symbols, strings, sentences). We refer to the set of all correct sequences as a language. When diagrams are involved, the situation becomes more complex: the syntax does not deal with the linear sequence of possible characters anymore, but must cope with two- or even three-dimensional diagrammatic elements, like different kinds of boxes (denoting states, classes, objects, etc.) and different kinds of arcs (denoting transitions, associations, links, etc.). In UML, the syntax is given by the UML meta-model. [23]

A.2 Semantics

In this section we focus on the concept of "semantics", introducing the semantic domain and the semantics mapping. A semantics definition consists of two parts: first a semantics domain is to be defined, and then a mapping from the syntax to the semantics domain is to be provided. [23]

A.2.1 Semantic Domain

The semantics of a language tells us about the meaning of each construct of the language. This is usually done by explaining new constructs of the language in terms of already known (and well understood) concepts. Thus a certain domain is needed that is assumed to be a well understood set of concepts. We call this domain the semantic domain, and denote it with S.

The semantic domain gives a notion of what concepts exist in the universe of discourse (UoD). A semantic domain can be called a "system model", as it describes our notion of what a system is. Thus, the semantic domain is an abstraction of reality, describing the important aspects of the systems that we are interested in developing. In UML the description of the semantic domain contains objects, values, messages that are passed around, etc. Unfortunately, this is informally explained and scattered throughout the whole UML description.

An underlying notation is necessary to describe the semantic domain S. Almost all general purpose formal languages (e.g. Z or Object Z), algebraic specification languages, and pure mathematics can be used. [23]

In practice, semantics and syntax are often mixed up, leading to considerable confusion. While syntax is the notation and therefore is what we manipulate, semantics tells us about the meaning of the notation. We do not directly manipulate semantics. Instead everything we see and work with on paper or electronically is a syntactic representation. Unfortunately this means that if we define "semantics", we also need a syntactical representation for it. Therefore, to some extent it is dependent on the purpose for which a language is used, to regard it as syntax or semantics.

A.2.2 Semantic Mapping

The third step of a semantics definition is to relate the syntactic concepts with the concepts of the semantic domain. The semantics definition is a mapping

$$M : L \to S$$

from the syntax L to the semantic domain S. Mathematics and graph transformations can be used to describe the semantic mapping.

A detailed example of the above concepts can be found in [16, 23].

Appendix B

Examples of the Supplemental Approach

B.1 Example of the Supplemental Approach, using LSL

This appendix presents an example illustrating the formalisation of a UML class diagram, using the Larch Shared Language (LSL). Figure B.1 presents this class diagram, which models a seminar scheduling system.

A seminar scheduling system maintains information about specific seminars, and about the people who will present these seminars. Each seminar is associated with a number of presenters. The presenters must be qualified for presenting a seminar. Each seminar is also associated with a number of offerings. Each offering specifies the original date of a seminar, as well as whether it has been cancelled or not. Each offering of a seminar is also associated with at most one presenter for that specific seminar. Finally, each offering of a seminar is associated with a number of attendees for that specific seminar.

Now we give the formalisation of this UML class diagram, as the LSL trait Seminar-Scheduling-System (this trait corresponds to the trait ClassTheory, described in Section 2.2.1).

$$
\begin{array}{l}
Seminar - Scheduling - System : trait \\
\quad includes \\
\qquad Type - SeminarSchedulingSystem, \\
\qquad Type - Presenter, Type - Seminar, \\
\qquad Type - Offering, Type - Attendee, \\
\qquad SeminarSchedulingSystem\_seminar, \\
\qquad SeminarSchedulingSystem\_presenter, \\
\qquad qualified\_qualifiedFor,
\end{array}
$$

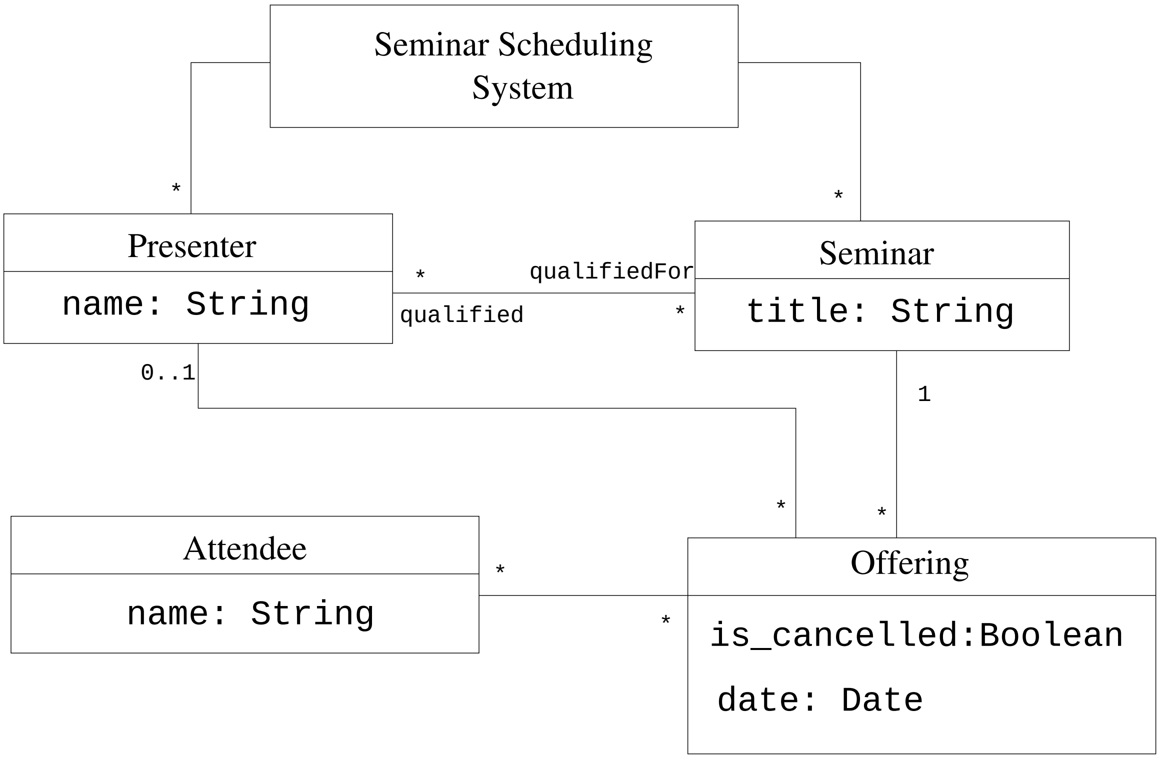

Figure B.1: Class Diagram for a Seminar Scheduling System (taken from [3]).

$presenter\_offering,$
$offering\_attendee,$
$offering\_seminar$
$introduces$
$asserts$
$implies$ **(possible invariants)**
The two roles of the association are derived from the set of pairs in its relation:

$\forall p : Type - Presenter, s : Type - Seminar$
$s \in qualifiedFor(p) = (p, s) \in qualified\_qualifiedFor$
$p \in qualified(s) = (p, s) \in qualified\_qualifiedFor$

Some cardinality constraints:

$\forall o : Type - Offering$
$card(Type - Seminar(o)) = 1$
$\forall o : Type - Offering$
$card(Type - Presenter(o)) \leq 1$

The included LSL traits *Type-SeminarSchedulingSystem, Type-Presenter, Type-Seminar, Type-Offering,* and *Type-Attendee* are the traits which specify the object types SeminarSchedulingSystem, Presenter, Seminar, Offering, and Attendee, respectively.

The included LSL traits with names $role1\_role2$ are the traits which specify associations with role names *role1* and *role2* between the corresponding object types. For example the trait *qualified_qualifiedFor* specifies the association between the object types Presenter and Seminar.

Some invariants on the whole class diagram have been expressed in the 'implies' section. The first two invariants state that whenever two objects of type Presenter and Seminar are related in an appropriate way, an association between the two objects must exist. The last two invariants express the constraints that each Offering must be associated with exactly one Seminar, and that each Offering must be associated with at most one Presenter.

In order to complete this example, it is necessary to give some of the LSL traits which formalise the object types and associations of Figure B.1. The object type Presenter is interpreted as a basic sort denoted by Presenter. Its attribute name is interpreted as a function name with signature

$$name : Presenter \rightarrow String$$

which is added to the specification for object type Presenter. The type String is interpreted as the sort of strings String, which is pre-defined in Larch. The other object types in Figure B.1 can be formalised in a similar way.

Associations between object types can be formalised as two related functions that map an object of one type to the set of associated objects of another type. Here we formalise the association between Presenter and Seminar, which has two role names *qualified* and *qualifiedFor*. This association can be represented as two functions qualified and qualifiedFor with the signatures:

$$qualified : Set[Seminar] \rightarrow Set[Presenter]$$
$$qualifiedFor : Set[Presenter] \rightarrow Set[Seminar]$$

where Set[Seminar] and Set[Presenter] are the power sorts of Seminar and Presenter. In addition, these two functions must satisfy the following axioms:

$$qualified(\{\}) = \{\}$$
$$qualifiedFor(\{\}) = \{\}$$
$$qualified(s \cup s') == qualified(s) \cup qualified(s')$$
$$qualifiedFor(s \cup s') == qualifiedFor(s) \cup qualifiedFor(s')$$

These axioms imply that these functions are completely determined by their values at singleton sets.

B.2 Example of the Supplemental Approach, using the RAL Theory

This appendix presents an example illustrating the formalisation of a UML class diagram, using the RAL theory. The UML class diagram to be formalised is the same one that was formalised in Section B.1.

A RAL theory representing this class diagram is:

$$
\begin{array}{l}
theory \\
\quad Seminar - Scheduling - System \\
types \\
\quad Presenter, Seminar, Attendee, Offering, SeminarSchedulingSystem \\
attributes \\
\quad \overline{Presenter} : F(Presenter) \\
\quad \overline{Seminar} : F(Seminar) \\
\quad \overline{Attendee} : F(Attendee) \\
\quad \overline{Offering} : F(Offering) \\
\quad \overline{SeminarSchedulingSystem} : F(SeminarSchedulingSystem) \\
\quad name : Presenter \rightarrow String \\
\quad title : Seminar \rightarrow String
\end{array}
$$

$is\_cancelled : Offering \rightarrow Boolean$
$date : Offering \rightarrow Date$
$qualified : Seminar \rightarrow F(Presenter)$
$qualifiedFor : Presenter \rightarrow F(Seminar)$
$\overline{qualified\_qualifiedFor} : Presenter <-> Seminar$
$\overline{SeminarSchedulingSystem\_Seminar} : SeminarSchedulingSystem <-> Seminar$
$\overline{SeminarSchedulingSystem\_Presenter} : SeminarSchedulingSystem <-> Presenter$
$\overline{Presenter\_Offering} : Presenter <-> Offering$
$\overline{Offering\_Attendee} : Offering <-> Attendee$
$\overline{Offering\_Seminar} : Offering <-> Seminar$

actions **(Standard predefined actions to modify classes and associations)**

$create_{Presenter}(p : Presenter)$
$kill_{Presenter}(p : Presenter)$
$create_{Seminar}(s : Seminar)$
$kill_{Seminar}(s : Seminar)$
$create_{Attendee}(a : Attendee)$
$kill_{Attendee}(a : Attendee)$
$create_{Offering}(o : Offering)$
$kill_{Offering}(o : Offering)$
$create_{SeminarSchedulingSystem}(se : SeminarSchedulingSystem)$
$kill_{SeminarSchedulingSystem}(se : SeminarSchedulingSystem)$
$add\_link_{qualified\_qualifiedFor}(s : Seminar, p : Presenter)$
$delete\_link_{qualified\_qualifiedFor}(s : Seminar, p : Presenter)$
$add\_link_{SeminarSchedulingSystem\_seminar}(se : SeminarSchedulingSystem, s : Seminar)$
$delete\_link_{SeminarSchedulingSystem\_seminar}(se : SeminarSchedulingSystem, s : Seminar)$
$add\_link_{SeminarSchedulingSystem\_presenter}(se : SeminarSchedulingSystem, p : Presenter)$
$delete\_link_{SeminarSchedulingSystem\_presenter}(se : SeminarSchedulingSystem, p : Presenter)$
$add\_link_{presenter\_offering}(p : Presenter, o : Offering)$
$delete\_link_{presenter\_offering}(p : Presenter, o : Offering)$
$add\_link_{offering\_attendee}(o : Offering, a : Attendee)$
$delete\_link_{offering\_attendee}(o : Offering, a : Attendee)$
$add\_link_{offering\_seminar}(o : Offering, s : Seminar)$
$delete\_link_{offering\_seminar}(o : Offering, s : Seminar)$

axioms **(Possible axioms)**

The two directions of the association are derived

from the set of pairs in its relation:
$\forall p : \overline{Presenter}; s : \overline{Seminar}$
$\quad s \in qualifiedFor(p) = (p, s) \in \overline{qualified\_qualifiedFor}$
$\quad p \in qualified(s) = (p, s) \in \overline{qualified\_qualifiedFor}$
Some cardinality constraints:
$\forall o : \overline{Offering}$
$\quad card(Seminar(o)) = 1$
$\forall o : \overline{Offering}$
$\quad card(Presenter(o)) \leq 1$

As described in Section 2.2.2, the 'types' section defines the names of all classes in the class diagram.

The 'attributes' section defines the attributes of each class, and all associations between classes. An association has the name $\overline{role1\_role2}$, and specifies an association with role names *role1* and *role2* between the corresponding object types. Also, for each class C there is an attribute $\overline{C} : F(C)$ denoting the set of existing instances of class C.

For each association, the 'actions' section defines actions *add_link* and *delete_link*. The 'actions' section also defines actions to create and kill each instance of a class.

The 'axioms' section defines constraints on the cardinality of each association end, as well as the property that whenever two objects of type Presenter and Seminar are related in an appropriate way an association between the two objects must exist. Note that the same properties were defined in Appendix B.1 using LSL.

Appendix C

Example of the OO-Extended Formal Language Approach

In this section we illustrate the OO-extended formal language approach, by giving an example of the formalisation of UML statecharts. Statecharts are used for behavioral description of classes. Classes with an assigned statechart are called reactive. Instances of reactive classes are called objects. The states of a statechart represent abstract situations in the life-cycle of objects.

C.1 Abstract Syntax and Static Semantics of UML Statecharts

As explained in Section 2.3, UML describes the abstract syntax of its metamodel using class diagrams. The UML class diagram shown in Figure C.1 describes the abstract syntax for a statechart of a reactive class and defines its execution. Our goal is to describe this abstract syntax using Object-Z classes and their state schemas, attributes and associations.

As shown in Figure C.1, Metaclass *StateMachine* consists of all the vertices and transitions of the statechart. Two special kinds of states are introduced in our metamodel: The initial state (metaclass *InitialState*) of a statechart represents the beginning of the object life-cycle, while the final state (metaclass *FinalState*) represents the end of the object life-cycle. The whole class diagram can be specified by the Object-Z class shown in Figure C.2.

In this Object-Z class specification, several consistency constraints have been formulated: The source vertices and target vertices of the statechart's transitions have to belong to the state machine. The entering and exiting transitions of all vertices of the state machine have to belong to the state machine, too. Initial and final states are direct substates of the state machine

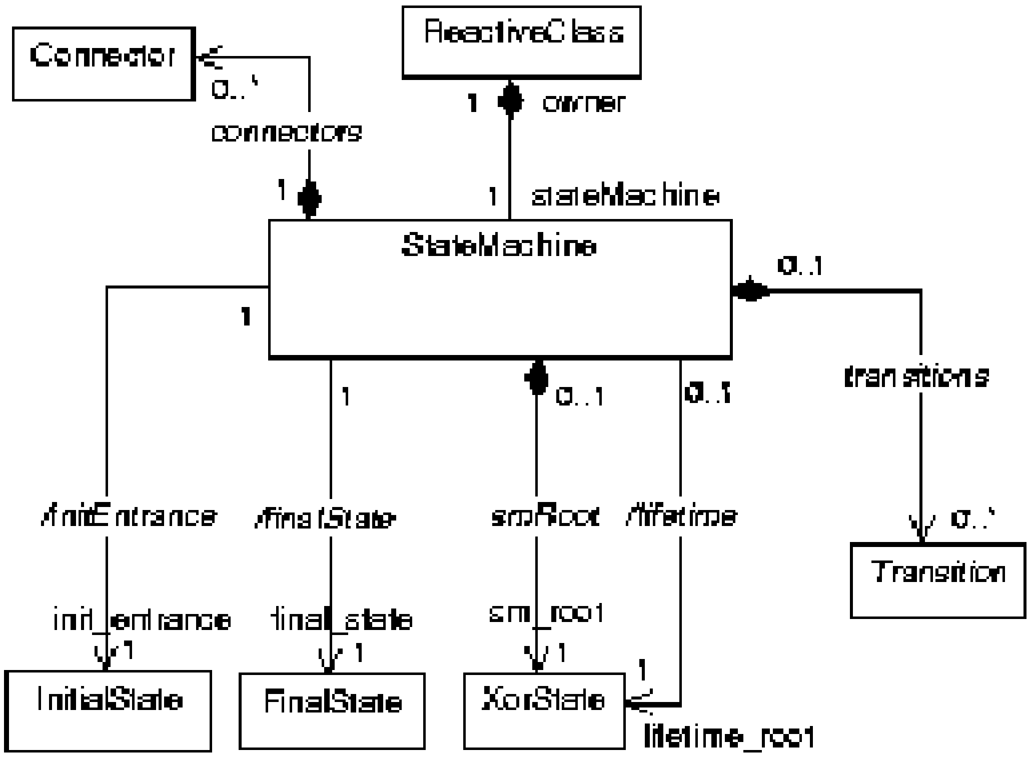

Figure C.1: Metaclass *StateMachine* (taken from [10]).

StateMachine

$$
\begin{array}{l}
owner : ReactiveClass \\
sm\_root : XorState_{\copyright} \\
transitions : \mathbb{F} \downarrow Transition_{\copyright} \\
connectors : \mathbb{F} \downarrow Connector_{\copyright} \\
\Delta \\
lifetime\_root : XorState \\
initial\_state : InitialState \\
final\_state : FinalState \\
\hline
sm\_root.subnodes = \\
\quad \{lifetime\_root, initial\_state, final\_state\} \\
sm\_root.en\_trans = sm\_root.ex\_trans = \varnothing \\
sm\_root.static\_reactions = \varnothing \\
sm\_root.entry\_action = sm\_root.exit\_action = \varnothing \\
lifetime\_root.substates^{\bullet} \cap \\
\quad (InitialState \cup FinalState) = \varnothing \\
\forall\, t : transitions;\ v : \downarrow Vertex \mid v \in (t.source \cup t.target) \bullet \\
\quad v \in sm\_root.subnodes^{+} \cup connectors \\
\forall\, v : \downarrow Vertex \mid v \in sm\_root.subnodes^{+} \cup connectors \bullet \\
\quad (v.en\_trans \cup v.ex\_trans) \subseteq transitions
\end{array}
$$

Figure C.2: Object-Z class specification for *StateMachine* (taken from [10]).

root state (sm_root).

C.2 Dynamic Semantics of UML Statecharts

As explained in Section 2.3, our goal is to describe the dynamic semantics for the UML metamodel using operations of the Object-Z classes. The dynamic semantics of statecharts defines how an object's statechart reacts to a received event. In order to define this, it is first necessary to define the "instantiation" of a state machine.

As shown in Figure C.3, the state machine instance of a reactive object manages the object-specific information for the statechart represented by a state machine: the active state configuration (metaclass StateConfiguration) and the history configuration (information about the last active state configurations.)

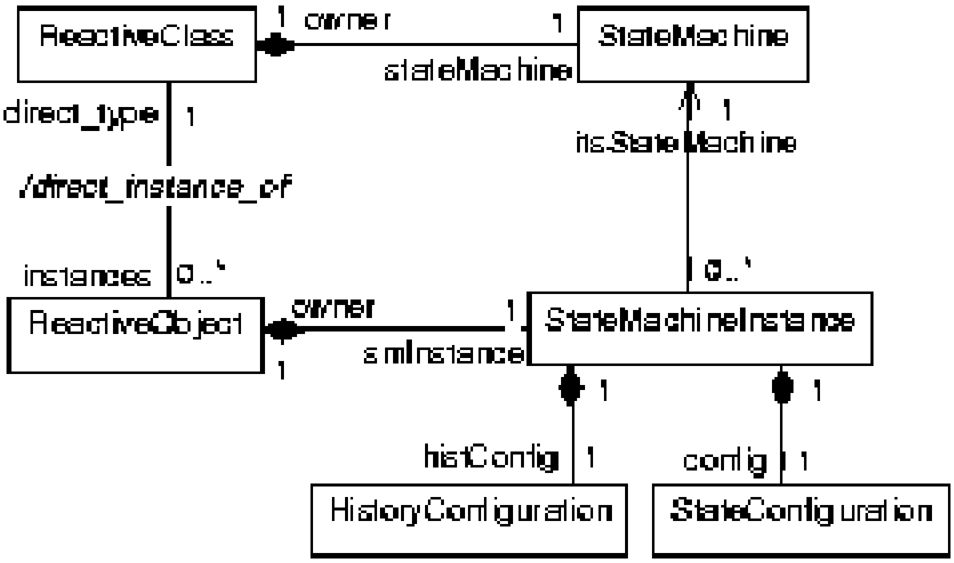

Figure C.3: State Machine Instance (taken from [10]).

Figure C.4 shows how this State Machine Instance can be represented in Object-Z.

Now we can define the reaction of a statechart to a received event. This reaction is triggered by the operation *DispatchEvent* of a state machine instance. As shown in Figure C.5, this operation builds an appropriate environment (the active state configuration, history configuration, received event) and then calls the operation *Execute* of its state machine. The operation *Execute* of the state machine defines the reactions to a received event.

Finally, it is necessary to briefly describe the task of the operation *Execute* of the metaclass *StateMachine*. The reaction of a statechart to a received event is defined on a step-by-step basis. Each step starts and finishes in a complete state configuration for the statechart. If in the resulting state configuration some of the states have outgoing transitions, a further step needs to be executed. The reaction is completed when the statechart reaches a state configuration where no state has an outgoing transition. The processing of a statechart's reaction to a received event has to be completed before the statechart can react to a new received event.

StateMachineInstance

$owner : ReactiveObject$
$itsStateMachine : StateMachine$
$config : StateConfiguration_{©}$
$histConfig : HistoryConfiguration_{©}$

---

details are omitted

Figure C.4: State Machine Instance specified using Object-Z (taken from [10]).

StateMachineInstance

$DispatchEvent \mathrel{\widehat{=}}$
$\quad BuildEnvironment \mathbin{⨾} itsStateMachine.Execute$

BuildEnvironment

$c! : StateConfiguration$
$h! : HistoryConfiguration$

$c! = config \land h! = histConfig$

Figure C.5: DispatchEvent Operation (taken from [10]).

Appendix D

Example of the Methods Integration Approach

This appendix illustrates the methods integration approach, by using it to prove that a property being asserted about a UML class diagram is true. Consider the class diagram D in Figure D.1, which describes the relationship between a university and its students. Given that full-time students are enlightened by a university, it is an interesting question to deduce the relationship between universities and students in general. This relationship is expressed by the class diagram D' at the bottom.

Lets imagine that a developer wants to use the methods integration approach, to prove that diagram D has the property that students in general are enlightened by a university. This corresponds to proving that diagrams D and D' are equivalent. Using a suitable sequence of transformation rules he/she should be able to transform the original diagram into the second diagram, thus proving that the derivation is valid. [14]

In this case, few steps are required to carry out the proof. One transformation rule allows the developer to move the "enlightens" association end from the "Full-time" subclass to its "Student" superclass, but requires that the opposite association end becomes optional. The second transformation rule permits the deletion of the "Full-time" and "Part-time" classes, as they are of no further interest. By applying correct transformations, diagram D gets transformed into diagram D', and the property is proved. [14]

Whenever a transformation rule is applied to a diagram, it must be shown that the resulting diagram is a valid deduction of the original diagram. For this to be shown, every syntactical modification of a UML diagram needs a semantic counterpart. [14] If the formal specification language is Z, then each diagrammatical transformation rule corresponds to the derivation of a new Z formula, as a result of applying an inference rule to the previous Z formula. Thus, each sequence of diagrammatical transformation rules corresponds to

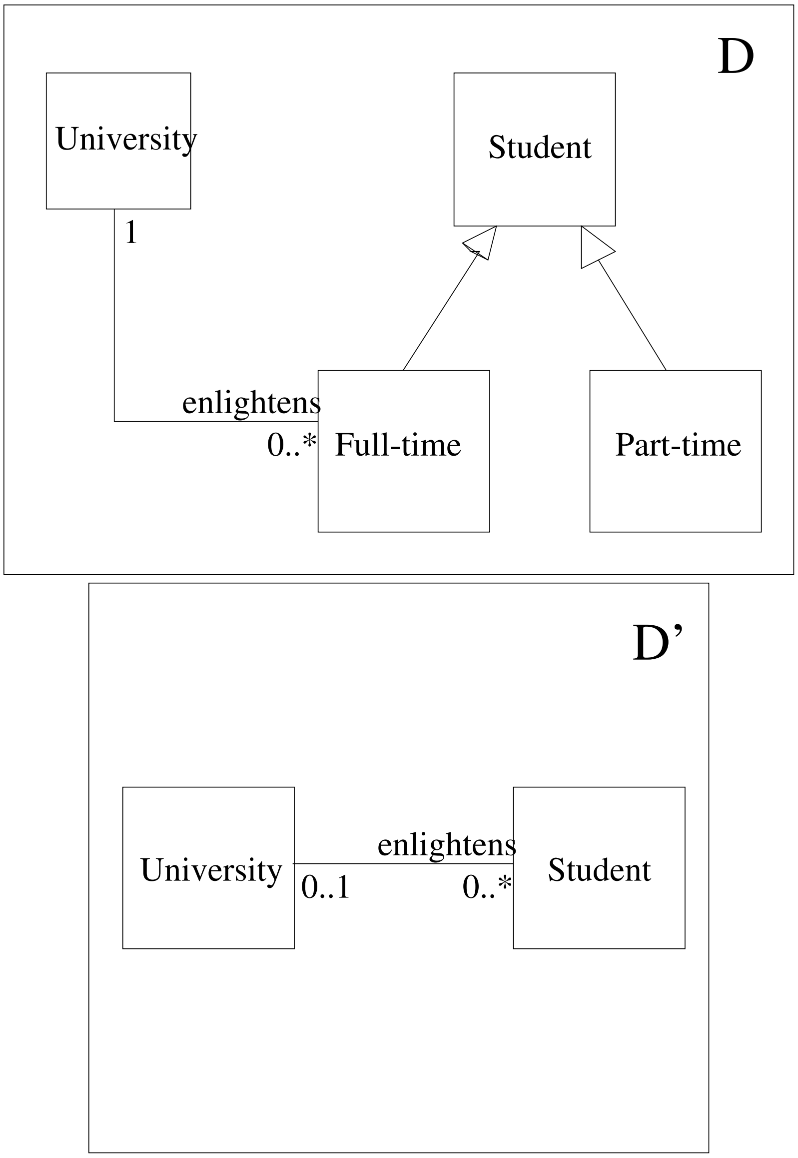

Figure D.1: Transforming a Class Diagram D to D' to derive information (taken from [16]).

a sequence of new Z formula derivations, by applying inference rules. This continues until the desired diagram is derived, and the property is proved.