Reasoning with UML Class Diagrams

Andy S. Evans
Department of Computer Science
University of York
Heslington
York, UK
andye@cs.york.ac.uk

Abstract

The Unified Modeling Language (UML) is likely to become an important industry standard language for modelling object-oriented systems. However, its use as a precise analysis tool is limited due to a lack of precise semantics and practical analysis techniques. This paper proposes a rigorous analysis technique for UML based on the use of diagrammatical transformations*. A precise description of a subset of UML class diagrams is presented. This is then used to identify a number of deductive transformations on class diagrams. Conditions for checking the soundness of the rules are also given. Because the reasoning system is based on the manipulation of diagrams, it is proposed that they can be successfully used by practitioners without recourse to complex linguistic proof techniques.*

1 Introduction

The popularity of object-oriented methods such as OMT [2], Booch [3] and Fusion [4] can be largely attributed to their use of visual, intuitively appealing, modelling notations. Typical examples of these notations are class and object diagrams, state diagrams and object-interaction diagrams. Each of these diagrams plays a role in presenting a particular view of the system being modelled.

It has long been argued that the effective use of graphical notations can be problematic when applied to the development of non-trivial systems. A significant source of problems is their lack of *precise semantics*. Most object-oriented methods only provide a loose interpretation of the meaning of the diagrams they use. This can lead to problems of: misinterpretation (confusion and disagreement over the precise meaning of a model); analysis (important properties of a model can only be informally verified) and design (correctness of designs cannot be checked).

These problems have been widely recognised [12], and have led to the development of a number of approaches to improving the precision of OO notations.

The most common approach to the problem has been to make the notations 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 [5, 6]). Most works focus on the generation of formal specifications from less formal OO models. This can reveal significant problems that are easily missed in less formal analyses of the models. Furthermore, the formal specifications produced can be rigorously analysed, providing another opportunity for uncovering problems. However, a serious limitation of this approach is that it requires an in-depth knowledge of the formal notation and its proof system. This can be a significant barrier to industrial use.

Another approach to the problem has been to extend formal notations with OO features, thus making them more compatible with OO notations. Several extensions exist in the literature (e.g., Z++ [7] and Object-Z [8]). However, although a rich body of formal systems have resulted, they are still too different from current industrial methods to be suitable for general industrial application. In addition, there is also a lack of available analysis tools.

Rather than generate formal specifications from informal OO models, a more workable approach might be to make the informal modelling constructs more *precise*. This approach allows developers to *directly manipulate* the OO models they have created. The role of formal specification techniques is therefore to gain an insight into appropriate semantics for informal modelling constructs[1].

This paper investigates the above approach in relation to the Unified Modeling Language (UML) [9]. It aims to show how rigorous reasoning techniques can be incorporated within UML at the level of its component abstrac-

[1]Further discussion on this approach can be found in [12]

tions and representations. Because these abstractions are presented using diagrams, we will investigate the use of *diagrammatical manipulation* as a proof technique for UML.

UML is a large modelling language, and therefore the proof techniques are only illustrated for a small part of it - the static model. In order to make the meaning of the model precise, a simple syntactic and semantic model is first constructed. Rules for manipulating a static model are then proposed based on diagrammatical transformations. These rules enable a diagram representing a static model of a system (a UML class diagram) to be transformed into a diagram representing some deduced property or conclusion about the system. It is shown how the soundness of the rules may be proven correct with respect to the semantic model. The end result is set of simple diagrammatical transformations, which provide a general foundation for understanding and verifying properties of UML class diagrams. Thus, it is shown that the use of diagrams within a rigorous development process is both feasible with and complementary to current software development practice.

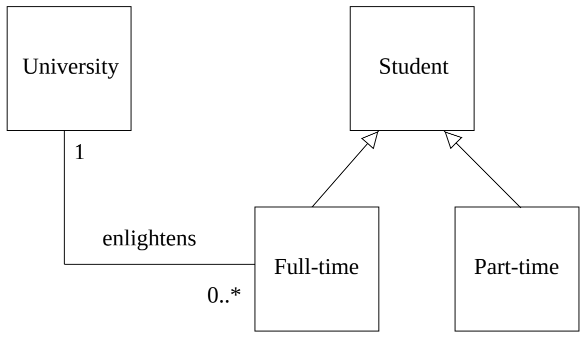
The paper is structured as follows: Section 2 gives a brief introduction to UML and UML class diagrams. In Sections 3 and 4 the well-formedness rules and semantics (of a subset of) UML class diagrams are precisely defined. Section 5 then presents rules for manipulating class diagrams in terms of diagrammatical transformations. Section 6 describes how the soundness of the transformation rules can be checked against the semantic model and a start is made to verify the soundness of a number of the rules. Finally, some related issues are discussed in Section 7 before concluding.

2 A brief introduction to UML and class diagrams

The Unified Modelling Language (UML)[9] is emerging as a de-facto standard for modelling object-oriented systems. It provides a visual language for modelling OO systems consisting of a number of diagram models. UML diagrams can be broadly divided up into static and behavioural diagrams. A static diagram describes the static (data) properties of a system, for example the relationships that hold between object instances over the system lifetime. A behavioural diagram describes the interactions that occur between objects in order to perform the functions of the system.

An important part of UML is its semantics document [10], which attempts to give a sound semantic basis to its diagrams. Meta-models are used to describe the syntax of its static and behavioural models, while semantic details are expressed in informal English. Unfortunately, the informal nature of these semantics is inadequate for justifying formal analysis techniques for UML. Thus, we need to develop a more precise semantic description. UML is also a large modelling language. Therefore, we restrict ourselves to considering analysis techniques for its *static model*. The static model is a core model in UML, and many components of a static analysis technique will apply to its behavioural models.

The static UML model is visually represented by a *class diagram*. Its purpose is to graphically depict the relationships holding among objects manipulated by a system. As an example, a typical class diagram is shown below, which depicts the relationship between a university and its students:



As this example shows, a UML class diagram provides a visually expressive and intuitive model of a system. However, it is less effective when it comes to answering important questions about the system it represents. In particular, it is not possible to reason (in a precise manner) with the diagram or deduce properties about it. For example, what is the relationship between the university and students? Furthermore, does a university enlighten any of its part-time students? Using informal arguments these questions might be answered like so: “some students must be enlightened by the university, as some students are full-time students”, or “clearly, there is no relationship between the university and part-time students - they are not connected”, or “surely, some part-time students can be enlightened - the class diagram does not forbid this”. In each case, a conjecture is made about the class diagram, but can it be rigorously verified? The answer is that it cannot, because the means to *prove* its correctness is missing.

It is these type of questions that this paper sets out to answer. By developing a precise description of what a UML class diagram means, we can develop sound rules for reasoning with UML models. Furthermore, it is hoped that the approach outlined in this paper will provide a basis for reasoning with other types of diagrams supported by UML.

Four fundamental aspects of UML class diagrams are dealt with in this paper:

- *classes*, which describe the different kinds of objects that can exist in a system;

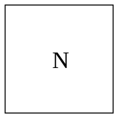
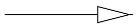
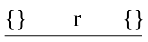
- *associations*, which describe what kinds of objects can be linked together;

- *multiplicity* constraints, which state how many objects can be related to each other;

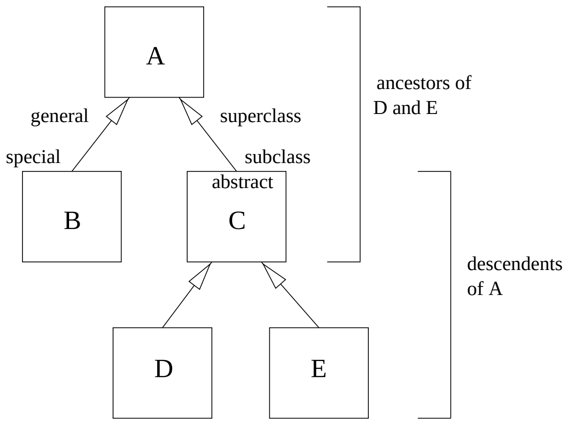
- *generalisation*, which is used to classify objects, and therefore simplify the overall structure of a design.

2.1 The components of UML class diagrams

A class diagram is composed of a number of primitive diagrammatic objects as follows:

Diagrammatic object	Name
	named class
	generalization
	named association (with multiplicity sets)

In general, a class diagram consists of a number of classes that are related (linked) by associations and generalisation hierarchies. Generalisation is a relationship between classes in which one class is identified as the *general* class and the others as *specialisations* of it. The generalisation relation is represented by arrowed line drawn from the specialised class to the general class. Classes may form a generalisation hierarchy. The *ancestors* of a class are all those classes found by traversing the hierarchy towards the top. Its *descendants* are those found by going downwards in the hierarchy from the class. Other terms used to describe the role a class plays within a particular generalisation relationship are: *superclass*, which denotes a generalising class and *subclass* which denotes a specialising class. Abstract classes (i.e. classes which act as a 'place-holder' for subclasses) are labelled as 'abstract'. The following diagram illustrates the general idea:



An association is a relationship between classes that specifies how instances of the classes are linked together. Note that an association may link (via its *association ends*) more than two classes. However, binary associations are the most common and will be used in this paper. Associations are annotated with additional information to show how many objects of the associated class an object can be related to. This multiplicity information is shown by a *set expression*, placed at each end of the association line. An association may also be *labelled* with a *name*, which uniquely identifies the association and *roles*, which denote the roles that instances.

3 Well-formed diagrams

In the UML semantics document (version 1.1), the core package - *relationships* - gives an abstract syntax for the static components of the UML (and therefore the abstract syntax of UML class diagrams). This is described at the meta-level using a class diagram with additional well-formedness rules given in the Object Constraint Language.

In this section we use the Z notation to precisely define the abstract syntax and well-formedness rules of a subset of UML class diagrams. We treat the UML semantics document as a *requirements statement* from which a fully formal model can be obtained.

The following given sets are assumed:

$$[\mathit{ClassName}, \mathit{Name}]$$

from which all class names and names can be drawn.

An association has association ends which describe the role that classes plays in the association:

```
┌─ AssociationEnd ─────────────────
│ rolename : Name
│ class : ClassName
│ multiplicity : ℙ₁ ℕ
├──────────────
│ multiplicity ≠ {0}
└──────────────────────────────────
```

An *AssociationEnd* has a *name* and is linked to a *class*. It also has a multiplicity describing the number of instances of the class that may participate in the association. A multiplicity must be a non-empty set of values and may not be a multiplicity of zero.

A binary association has a name and two association ends:

$$\begin{array}{|l}
\textit{Association} \\
\hline
\textit{name} : \textit{Name} \\
e_1, e_2 : \textit{AssociationEnd} \\
\hline
e_1.\textit{rolename} \neq e_2.\textit{rolename} \\
\hline
\end{array}$$

The constraint of the schema states that each *AssociationEnd* of an *Association* must have a unique rolename.

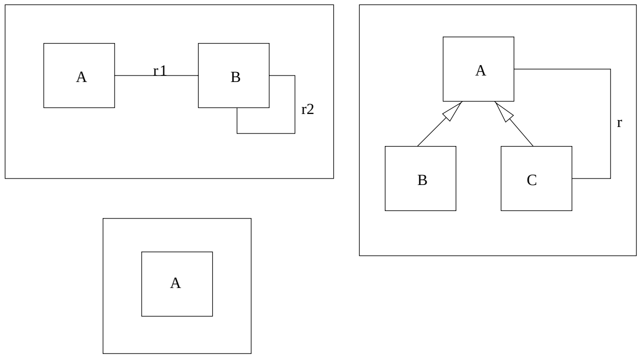
A schema describing the objects in a class diagram and the rules by which they are well-formed can now be given. A well formed diagram consists of a set of classes, a set of abstract classes, a set of associations and a superclass relationship between classes:

$$\begin{array}{|l}
\textit{WFD} \\
\hline
\textit{abstract}, \textit{classes} : \mathbb{F}\,\textit{ClassName} \\
\textit{associations} : \mathbb{F}\,\textit{Association} \\
\textit{superclass} : \textit{ClassName} \nrightarrow \textit{ClassName} \\
\textit{allsup}, \textit{allsub} : \textit{ClassName} \nrightarrow \mathbb{F}\,\textit{ClassName} \\
\hline
\textit{abstract} \subseteq \textit{classes} \\
\forall\, a_1, a_2 : \textit{associations} \mid \\
\qquad a_1 \neq a_2 \bullet a_1.\textit{name} \neq a_2.\textit{name} \\
\forall\, c : \textit{classes} \bullet c \notin \textit{allsub}(c) \land c \notin \textit{allsup}(c) \\
\exists\, r : (\textit{classes} \leftrightarrow \textit{classes}) \bullet \textit{superclass} \cup \\
\qquad \{a : \textit{associations} \mid \\
\qquad\qquad a.e_1 \neq a.e_2 \bullet a.e_1.\textit{class} \mapsto a.e_2.\textit{class}\} = r^* \\
\hline
\end{array}$$

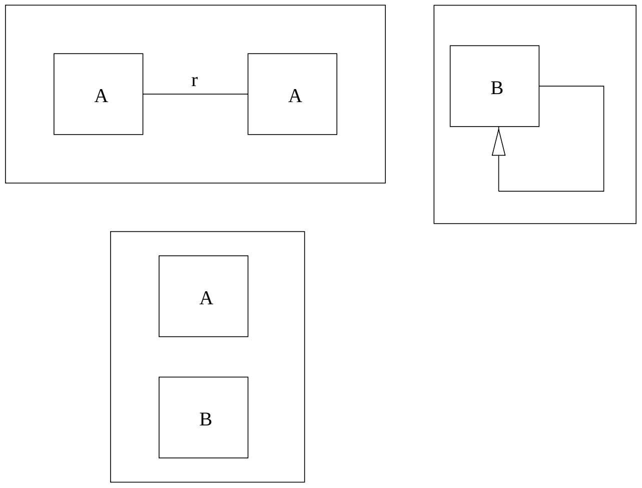
The constraints of the schema state that:

- some classes may be abstract;
- all associations have unique names;
- no class can be supertype of its subclasses, circular inheritance is not allowed;
- a transitive, reflexive relationship exists between all classes in a diagram.

where *allsub* and *allsup* are un-specified functions, which return a given class's subclasses and superclasses respectively.
Here are some examples of diagrams that are well formed:



and some that are not:



4 Semantics

When drawing a diagram to represent a system, we intuitively learn to assign a meaning to its representational components. In the case of a class diagram, a class represents a set of objects. Additional *constraints* are placed on these objects by the relationships that exists between classes. For example, a part-time student class represents a set of part-time student objects. An enlightens association represents a set of links between university and part-time student objects and thus constrains their relationship to one another. Thus, a semantic model for class diagrams should describe the valid sets of objects that can be assigned to each of its classes and the constraints that are enforced upon them.

4.1 Set assignments

When a class diagram is drawn, we make the following assumptions regarding its *meaning*:

- A class is associated with a set of object instances;

- Each instance has an *identity* which distinguishes it from all other instances;

- An association represents a set of links between instances of classes.

The following data type is introduced from which the set of all object identities and object links can be drawn:

$$[OId, Link]$$

The following schema describes the set of instances and links that can be assigned to a class diagram:

$$\begin{array}{l} S \\ \hline obj : ClassName \twoheadrightarrow\!\!\!\!\!\!\shortmid\;\; \mathbb{P}\, OId \\ links : Name \twoheadrightarrow\!\!\!\!\!\!\shortmid\;\; (OId \leftrightarrow OId) \\ \hline \end{array}$$

4.2 Satisfaction conditions

We now formalise what it means for a set assignment to *satisfy* the components of a class diagram. A diagram component can be a class, a generalisation or an association. In addition, it is necessary to distinguish between *abstract* and *nonabstract* classes and their subclasses. We also want to be able to say what it means for a set assignment to satisfy a whole diagram, so a well-formed diagram must also be defined.

The enumerated type *Component* is introduced to represent the different components of a class diagram:

$$\begin{array}{l} Component ::= \\ \quad class\langle\langle ClassName\rangle\rangle \mid \\ \quad gen\langle\langle ClassName \times ClassName\rangle\rangle \mid \\ \quad abstract\langle\langle ClassName \times \mathbb{F}\, ClassName\rangle\rangle \mid \\ \quad nonabstract\langle\langle ClassName \times \mathbb{F}\, ClassName\rangle\rangle \mid \\ \quad association\langle\langle Association\rangle\rangle \mid \\ \quad wfd\langle\langle WFD\rangle\rangle \end{array}$$

The following relation $\models$ describes precisely what it means for an set assignment to satisfy each of the components of a class diagram.

4.2.1 Class

$$\begin{array}{l} \_ \models \_ : S \leftrightarrow Component \\ \hline \forall\, s : S;\; c : ClassName \bullet \\ \quad s \models class(c) \Leftrightarrow \\ \quad\quad c \in \operatorname{dom} s.obj \end{array}$$

A set assignment satisfies a class if objects may be assigned to the class.

4.2.2 Generalisation

$$\begin{array}{l} \forall\, s : S;\; sub, sup : ClassName \bullet \\ \quad s \models gen(sub, sup) \Leftrightarrow \\ \quad\quad s.obj(sub) \subseteq s.obj(sup) \end{array}$$

A set assignment satisfies a generalisation if the set of objects assigned to the subclass of the generalisation is a subset of those assigned to the superclass of the generalisation.

Conceptually, this models the fact that instances of a specialised class are also instances of its generalised class. Thus, whenever an instance of a specialised class is created, it also thought of as being an instance of all its ancestor classes. This approach (see France et al. [13]) also enables multiple-inheritance to be modelled (multiple inheritance implies membership of an instance to multiple classes) and also abstract and non-abstract classes (see sections 4.2.3 and 4.2.4).

4.2.3 Abstract Classes

$$\begin{array}{l} \forall\, s : S;\; abs : ClassName;\; subs : \mathbb{F}\, ClassName \bullet \\ \quad s \models abstract(abs, subs) \Leftrightarrow \\ \quad\quad (\exists\, x : \operatorname{seq}(\mathbb{P}\, OId) \mid \\ \quad\quad\quad \operatorname{ran} x = \{c : subs \bullet s.obj(c)\} \bullet \\ \quad\quad\quad\quad x \text{ partition } s.obj(abs)) \end{array}$$

A set assignment satisfies an abstract class if the set of objects assigned to its subclasses partition the set of objects assigned to the abstract class. Thus, all instances of an abstract class must belong to its subclasses.

4.2.4 Non-abstract classes

$$\begin{array}{l} \forall\, s : S;\; non : ClassName;\; subs : \mathbb{F}\, ClassName \bullet \\ \quad s \models nonabstract(non, subs) \Leftrightarrow \\ \quad\quad (\exists\, x : \operatorname{seq}(\mathbb{P}\, OId) \mid \\ \quad\quad\quad \operatorname{ran} x = \{c : subs \bullet s.obj(c)\} \bullet \\ \quad\quad\quad\quad \text{disjoint } x) \end{array}$$

A set assignment satisfies a non-abstract class if the set of objects assigned to its subclasses are disjoint. In this case, an object may be an instance of an non-abstract class or its subclasses.

4.2.5 Associations

$$
\begin{array}{l}
\forall s : S;\ a : Association \bullet \\
\quad s \models association(a) \Leftrightarrow \\
\quad\quad \mathrm{dom}(s.links(a.name)) \subseteq s.obj(a.e_1.class) \wedge \\
\quad\quad \mathrm{ran}(s.links(a.name)) \subseteq s.obj(a.e_2.class) \wedge \\
\quad\quad (\forall i : s.obj(a.e_1.class) \bullet \\
\quad\quad\quad \#\{j : s.obj(a.e_2.class) \mid \\
\quad\quad\quad\quad (i,j) \in s.links(a.name)\} \in \\
\quad\quad\quad\quad\quad a.e_2.multiplicity) \wedge \\
\quad\quad (\forall j : s.obj(a.e_2.class) \bullet \\
\quad\quad\quad \#\{i : s.obj(a.e_1.class) \mid \\
\quad\quad\quad\quad (i,j) \in s.links(a.name)\} \in \\
\quad\quad\quad\quad\quad a.e_1.multiplicity)
\end{array}
$$

A set assignment satisfies an association between classes if the following conditions holds:

- an association only links objects belonging to the classes it associates;
- each instance of a class attached to a role e_1 is linked (by the association) to e_2*.multiplicity* instances of the class attached to e_2 (and vice versa).

This definition implies a number of obvious properties. Given a named association a, attached to named classes c_1, c_2, with multiplicities m_1 and m_2, then:

- $0 \notin m_2 \Rightarrow \mathrm{dom}\, s.links(a) = s.obj(c_1)$, i.e. if there is a compulsory association at class c_2 then every instance of c_1 must be linked to an instance of c_2;
- $0 \in m_2 \Rightarrow \mathrm{dom}\, s.links(a) \subseteq s.obj(c_1)$, i.e. if there is a non-compulsory association at class c_2 then a subset of instances of c_1 are linked to an instance of c_2.

4.2.6 Diagrams

$$
\begin{array}{l}
\forall s : S;\ d : WFD \bullet \\
s \models wfd(d) \Leftrightarrow \\
\quad (\forall c : d.classes \bullet \\
\quad s \models class(c)) \wedge \\
\quad (\forall g : d.superclass \bullet \\
\quad s \models gen(first\, g, second\, g)) \wedge \\
\quad (\forall c : d.abstract \bullet \\
\quad s \models abstract(c, \mathrm{dom}(d.superclass \rhd \{c\}))) \wedge \\
\quad (\forall c : d.classes \mid c \notin d.abstract \bullet \\
\quad s \models nonabstract(c, \mathrm{dom}(d.superclass \rhd \{c\}))) \wedge \\
\quad (\forall a : d.associations \bullet \\
\quad s \models association(a))
\end{array}
$$

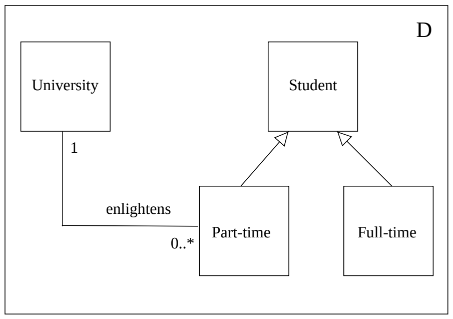
Finally, a set assignment satisfies a well-formed diagram d if and only if it satisfies all the components of d.

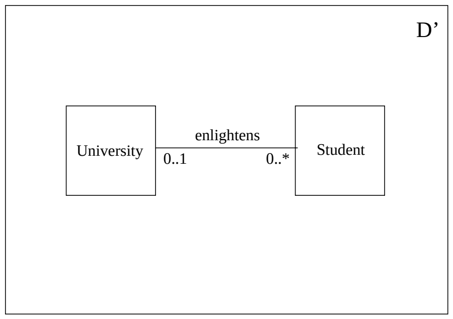
5 Reasoning with UML Diagrams

We now describe how UML class diagrams can be manipulated so that certain properties of one diagram to be deduced from another The means by which this is achieved is to present a set of *transformation rules* on UML diagrams. Each rule describes the legal transformations that can be applied to a class diagram (or other UML diagram), which result in the transformed diagram being a valid deduction of the original diagram.

As an example, consider the class diagram presented in Section 2, which describes the relationship between a university and its students. One of the conjectures made about the diagram was that it could be deduced that "some students must be enlightened by the university".

This *conjecture* can be expressed by the following two diagrams, where D is the diagram representing the original assumptions made about the model and D' represents the proposed conclusion:





Using suitable transformation rules, we should be able to transform the original diagram into the second diagram, thereby proving that the conjecture is a valid theorem.

However, before developing these rules, we must first describe what it means for one class diagram to *follow* from other class diagrams. In other words, when is it true that one class diagram is a valid consequence of another class diagram?

The consequence relation among class diagrams is defined as follows:

$$\begin{array}{|l} \_ \models_d \_ : WFD \leftrightarrow WFD \\ \hline \forall D, D' : WFD \bullet \\ \quad D \models_d D' \Leftrightarrow \\ \qquad (\forall s : S \bullet s \models wfd(D) \Rightarrow s \models wfd(D')) \end{array}$$

A well formed diagram D' follows from a well formed diagram D, if and only if, every set assignment that satisfies D also satisfies D'.

As described above, we wish to deduce properties of class diagrams by applying transformation rules. A transformation rule is valid provided that it allows us to transform a diagram D to D', and $D \models_d D'$. If this can be shown to be true for all transformation rules, then we can claim that a diagram D' resulting from the application of a transformation rule is a valid deduction of the original diagram D and therefore $D \vdash D'$.

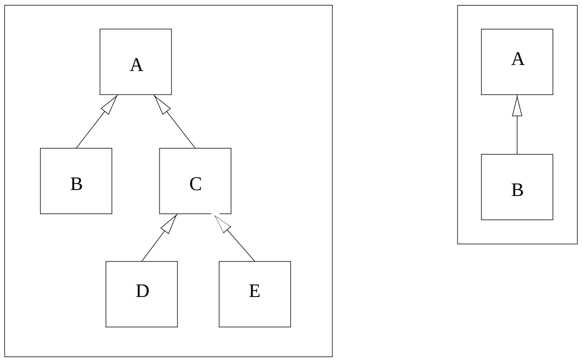
5.1 Deductive Transformation Rules

A set of deductive transformation rules are now presented for transforming UML class diagrams in which the transformed diagram is a deductive consequence of the original diagram. Each rule aims to identify simple transformations that can be applied in the proof of a number of class diagram properties.

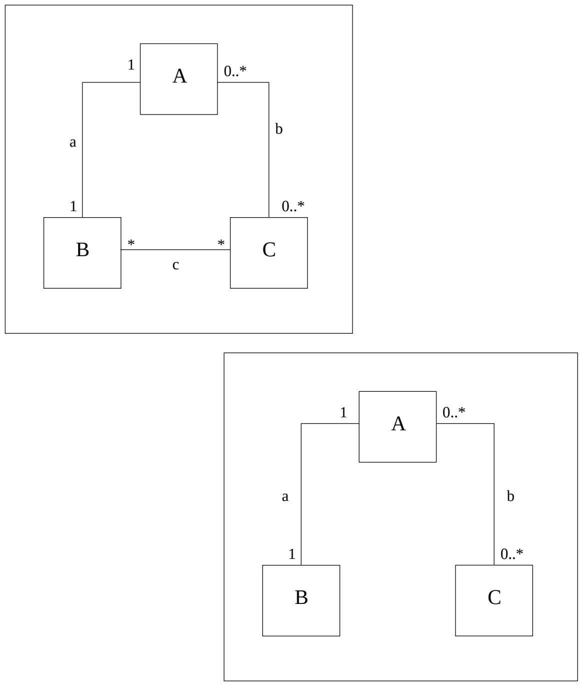
Rule 1: The rule of erasure of a diagrammatic object

A diagram may be copied omitting a class, association or generalisation. A subclass of an abstract class cannot be omitted. A well-formed diagram must result.

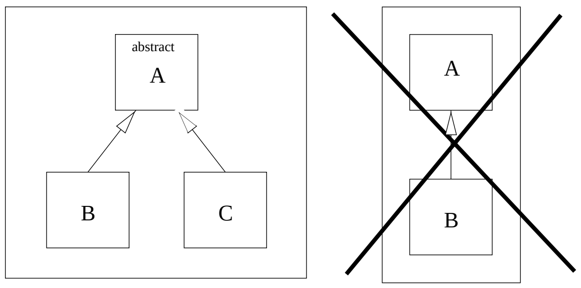
Example 1a *We are allowed to transform the diagram on the left to the diagram on the right by applying the erase class rule three times in the order: $D \rightarrow E \rightarrow C$*



Example 1b *We are allowed to transform the diagram on the left to the diagram on the right by erasing the association r:*



Example 1c *The following transformation is not permitted as a subclass of an abstract type is deleted:*

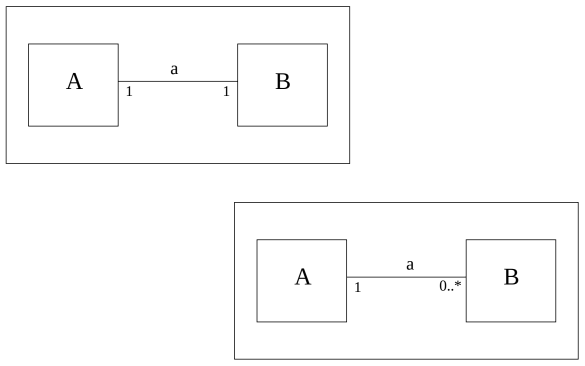


Rule 2: The rule of substitution of an association

An association may be substituted with a less constrained association of the same name as follows:

- in any association, R, an association end E with multiplicity M may be substituted with an association end E with multiplicity N provided that $M \subseteq N$

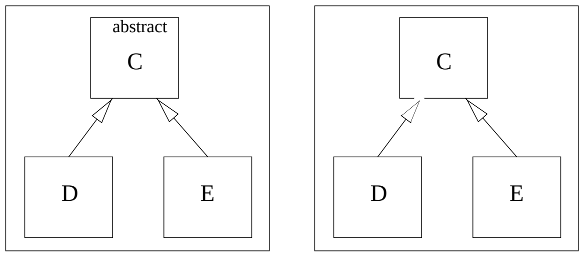
Example 2a *In the following, the diagram on the left is transformed to the diagram on the right substituting a less constrained association end:*



Rule 3: The rule of substitution of a class

A non-abstract class may be substituted for an abstract class of the same name.

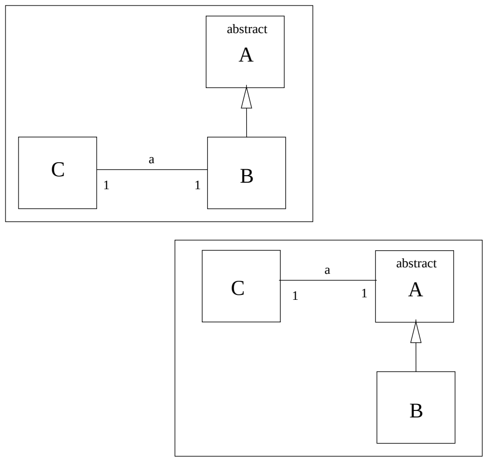
Example 3 *The diagram on the left is transformed to the diagram on the right substituting a non-abstract class for the abstract class:*



Rule 4: The rule of promotion of an association

Consider an association R with multiplicity M_1 (connected to a class C_1) and multiplicity M_2 (connected to class C_2). If C_2 is a subclass, then R may be 'promoted' to the superclass of C_2 provided that its multiplicity after the transformation at C_1 is optional, i.e. $0 \in M_1$.

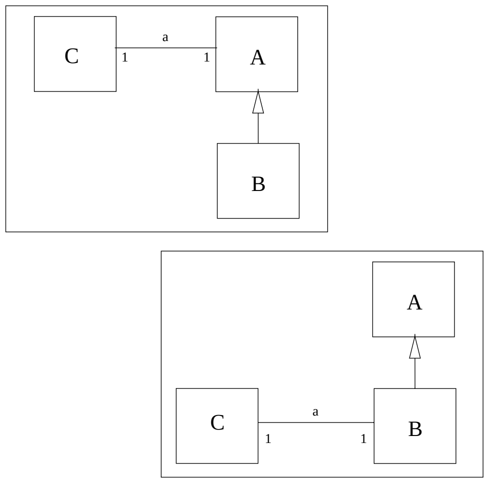
Example 4 *The diagram on the left is transformed to the diagram on the right by promoting the association from B to C:*



Rule 5: The rule of demotion of an association

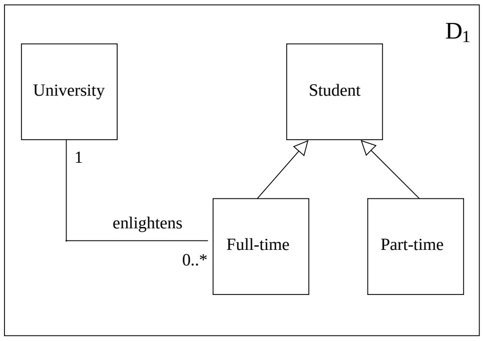
An association may be demoted from a superclass to a subclass provided that the superclass is abstract and it has only one subclass.

Example 5*The diagram on the left is transformed to the diagram on the right by demoting the association from A to B:*

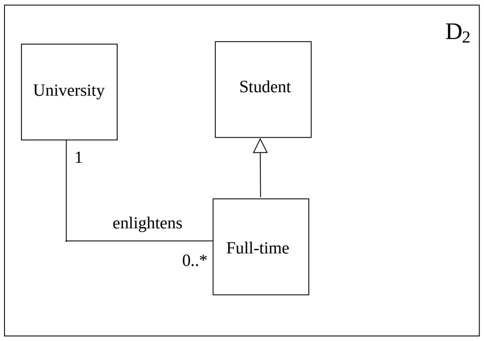


5.2 Example revisited

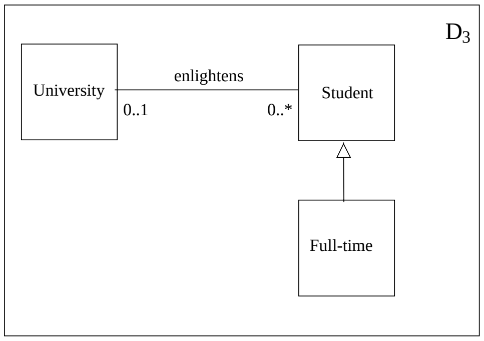
We now prove the class diagram conjecture in Section 5. Given the original model:



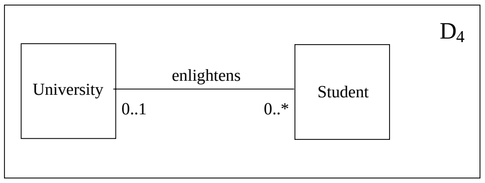
First, rule 1 is used to erase the *Part ⇔time* class:



Next, rule 4 is used to promote *enlightens* to the *Student* class. Note, that the multiplicity of the association at the *University* class is now optional:



Finally, the *Full ⇔time* class is erased (rule 1) leaving the desired conclusion (only some Students are enlightened by a University):



6 Soundness of Rules

See Appendix A

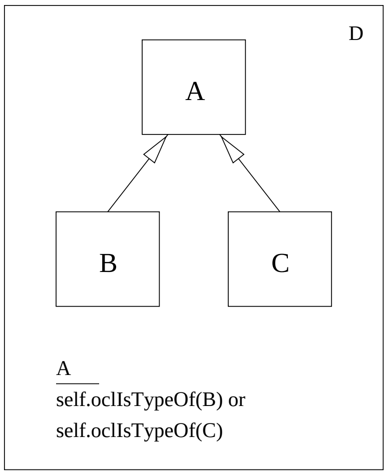
7 Issues

7.1 Constraints

Unfortunately, class diagrams have limited representational power. For example, they cannot describe dependencies between attributes of classes. Therefore, it is necessary to combine class diagrams with a textual language for describing constraints. In UML, such constraints can be added using the Object Constraint Language (OCL), a semi-formal language which uses set theory and predicate logic to describe constraints on class diagrams)[2].

In order to represent non-trivial designs it is likely that additional constraints and properties will have to be described using a language like OCL. However, will the work presented in this paper still be useful? Although we presently have limited practical experience of proving properties of large (non-trivial) class diagrams, we believe that the rules outlined in this paper are still applicable. This is because, in practice, textual constraints are usually applied to attributes of classes, and therefore will not constrain properties of association and generalisation constraints.

Nevertheless, constraints can be applied to class diagrams which invalidate a particular rule. For example, consider the following class diagram and OCL constraint:



```
A
self.oclIsTypeOf(B) or
self.oclIsTypeOf(C)
```

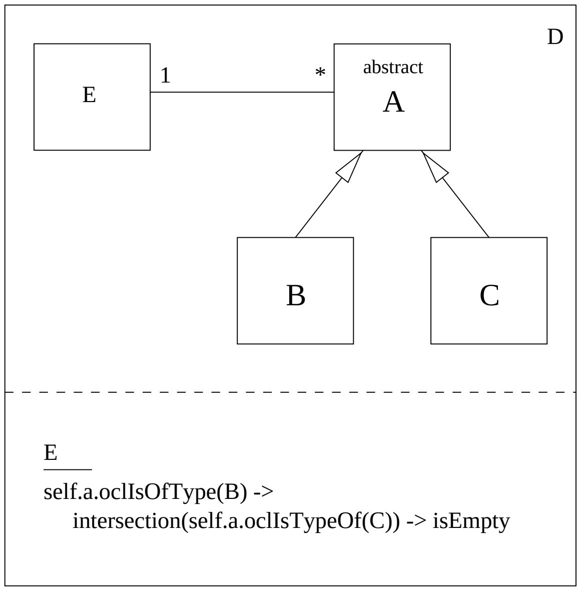
Here the OCL expression constrains every instance of *A* to be of type *B* or *C*. In this case, applying rule 1 to erase either class *B* or *C* will no longer be a valid transformation, as the constraint does not imply that every instance of *A* is of type *B* alone (or *C* alone).

[2]Further details of OCL can be found at http://www.software.ibm.com/ad/ocl/

In practice, it is unlikely that such a constraint would be defined, as the property is more elegantly described by an abstract class. However, care must be taken to ensure that a particular textual constraint does not interfere with the validity of a transformation.

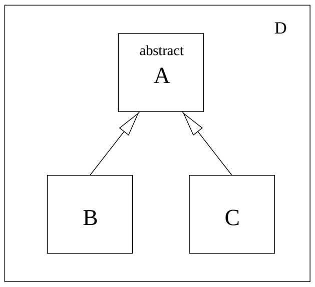
7.2 Proof patterns

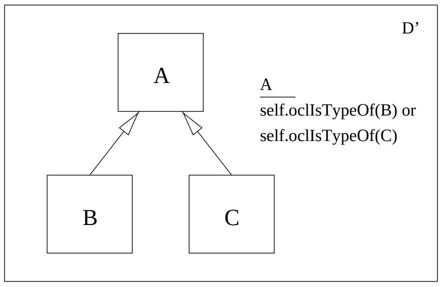
The idea that design patterns [14] can benefit the development process is becoming an accepted one. However, the ability to diagrammatically construct precise proofs of UML diagrams leads to the possibility of *proof and equivalence patterns*. A proof pattern encompasses the identification of a common pattern of proof, for instance: the verification of an invariant property, the proof of validity of a refinement, or perhaps the identification of commonly derived properties of a UML diagram. As a simple example, consider the following *proof pattern diagram*:



This diagram aims to document in an abstract manner a commonly inferred property of a class diagram, in which a class *E* is associated with a superclass. The derived property of this class diagram is that the sets of objects of type *B* associated with an instance of *E* is disjoint from the set of objects of type *C*.

In contrast, an equivalence pattern would document common equivalences between UML diagrams, for example:





Here diagram D is equivalent to diagram D' as the constraints of D' imply the implicit properties of the abstract class in diagram D (and vice versa).

In fact, any of the transformation rules presented in this paper can be thought of as proof patterns (albeit the most simple). The considerations of using a constraint language should also result in interesting proof patterns, but again these are best determined from practical application.

8 Conclusions

The diagrams used by UML and other OO methods are intuitive and understandable to practitioners in industry. Moreover, the abstractions they represent are designed to be close to those in the software domain. Yet, they do not offer the precision that is often required when reasoning with software systems. To achieve this goal, a full understanding of the laws by which they can be manipulated and reasoned with must be obtained.

This paper has a proposed and illustrated a step towards this goal. A formal semantics was developed for a small subset of the language of UML class diagrams. These semantics were then used as a basis for developing deductive transformation rules. Because these rules were based on diagrammatical transformations, it is hoped that a more intuitive approach to proof will result.

A number of extensions to the work presented here are currently being investigated (in addition to those described above): Firstly, it would be very useful to be able to prove the completeness of the rules. This would require showing that any valid static diagram is reachable by application of the rules from any other diagram. Secondly, the work presented here needs to be expanded to include other UML diagrams, particularly behavioural diagrams such as sequence diagrams and object interaction diagrams. An understanding of the relationship between the different diagrams supported by UML will be required in order to determine the effect that transformation rules will have on linked diagrams. For example, if it can be deduced that an object instance *a* of class *A* can send messages to many different instances of class *B* (on an object interaction diagram), then we should be able to deduce a one to many association between *A* and *B* (on a class diagram).

8.1 Related work

A significant amount of research has already been carried in the area of formal object-oriented methods. In addition to that referenced in the introduction, Lano et al. have investigated and formalised many aspects of the Syntropy method [15, 16]. This has involved showing how the diagrams used in Syntropy (including static diagrams) can be formalised using the Object Calculus. Lano has also proposed the idea of formalising design patterns as transformations on object diagrams. He has also verified a number of these patterns using the Object Calculus. The main differences between this work and theirs is that ours has concentrated on deductive transformations. Also, we have only considered a subset of class diagram components with the aim of identifying a complete set of basic transformations for their manipulation.

9 Acknowledgements

Tony Clark, Ian Hayes and members of the pUML group are thanked for their suggestions and comments on earlier versions of this work.

References

[1] Folwer M., UML Distilled, Addison-Wesley, 1997.

[2] Rumbaugh T. et al., Object-Oriented Modeling and Design, Prentice Hall, 1991.

[3] Booch, G., Object-Oriented Analysis and Design with Applications (2nd Ed), Benjamin Cummings, 1994.

[4] Coleman, et al., Object-Oriented Development: The Fusion Method, Prentice Hall, 1994.

[5] Bruel, J.M., France, B and Larrondo-Petrir, M., CASE-based Rigorous Object-Oriented Methods, In A.S.Evans and D.J.Duke, Procs of the 1st Northern Formal Methods Workshop, Springer eWiC series, 1997.

[6] Randolph Johnson, D. and Kilov, H., Can a flat notation be used to specify an OO system: using Z to describe RM-ODP constructs, In Elie Najm and Jen-Bernard Stephani, editors, Procs of the 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, pages 407-418, Chapman and Hall, 1996.

[7] Lano, K. and Haughton, H., The Z++ Manual. Technical Report, Imperial College, 1994.

[8] Duke, D., Object-Oriented Formal Specification, PhD thesis, University of Queensland, 1991.

[9] Booch, G., Jacobson, C., and Rumbaugh, J., The Unified Modeling Language - a reference manual, Addison Wesley, 1998.

[10] Booch, G., Jacobson, C., and Rumbaugh, J., The Unified Modeling Language Semantics Document (version 1.1), available from http://www.rational.com/uml, 1998.

[11] Spivey, J.M., The Z Notation - a reference manual, Prentice Hall, 2nd Edition, 1992.

[12] France, R., Evans A., Lano K., and Rumpe B., The UML as a Formal Modeling Notation, Computer Standards and Interfaces, No. 19, pages 325-334, 1998.

[13] France, R., J-M. Bruel, and M.M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modelling Environment, Journal of Object-Oriented Programming, To appear.

[14] Gamma, E., Helm, H., Johnson R., Vlissides, J., Design Patterns: Elements of Resuable Object-Oriented Software. Addison Wesley. 1994.

[15] Lano, K., Formalizing Design Patterns, Procs. of 1st Northern Formal Methods Workshop, Springer eWiC series, 1996.

[16] Lano K., Sanchez, A., Transformation Formal Development of Real-time systems, Transformation-based Reactive Systems Development, M Bertran, T. Rus (Eds), LNCS vol 1231, Springer-Verlag, 1997.

Appendix A

In Section 5 it was shown what is means for a diagram to be a logical consequence of another. In this appendix, the validity (soundness) of some of the transformation rules are proved by showing that each results in a diagram that is a logical consequence of the other.

9.1 Proof of the rule of erasing a class

Recap, there are two cases where a class can be omitted from a class diagram:

1. Where it is a subclass of a non-abstract class

2. Where it is related to another class by and association (in this case, the association must be deleted as well in order to preserve the well-formedness of the diagram)

It must be shown $D \models_d D'$ for each of the above cases.

Case (1): consider a non-abstract hierarchy with n subclasses $sub_1 \,..\, sub_n$ and non-abstract superclass sup. The set of objects assigned to each subclass is a subset of those assigned to sup (consequence of $s \models gen(sub_{x \in n}, sup)$) and is disjoint (consequence of $s \models non \Leftrightarrow abstract(sup, sub_1 \,..\, sub_n)$). Deletion of a subclass will be valid if it can be shown that the objects assigned to $n \Leftrightarrow 1$ subclasses is also disjoint, i.e:

$$
\begin{array}{l}
\forall\, s : S \bullet \\
\quad (s.obj(sub_1) \subseteq s.obj(sup) \wedge \quad .. \\
\quad s.obj(sub_n) \subseteq s.obj(sup) \wedge \\
\quad s.obj(sub_1) \cap .. \cap s.obj(sub_n) = \emptyset) \qquad \text{[disjoint]} \\
\quad \Rightarrow \\
\quad (s.obj(sub_1) \subseteq s.obj(sup) \wedge \quad .. \\
\quad s.obj(sub_{n-1}) \subseteq s.obj(sup) \wedge \\
\quad s.obj(sub_1) \cap .. \cap s.obj(sub_{n-1}) = \emptyset)
\end{array}
$$

which clearly holds as $P \cap Q \cap R = \emptyset \Rightarrow P \cap Q = \emptyset$, and $P \subseteq S, Q \subseteq S \Rightarrow P \subseteq S$.

Case (2): this is trivially true, as, given a diagram with classes C_1, C_2, related by an association R, it is always possible to deduce the existence of C_1 (provided that the resulting diagram is well-formed) without R and C_2:

$$
\begin{array}{l}
\forall\, s : S \bullet \\
\quad s \models class(C_1) \wedge \\
\quad s \models class(C_2) \wedge \\
\quad s \models association(R) \Rightarrow \\
\quad s \models class(C_1)
\end{array}
$$

Finally, it is shown that a subclass of an abstract class cannot be validly erased. In addition to the properties for a non-abstract class above, the following would need to be true if the transformation is permitted:

$$
\begin{array}{l}
\forall\, s : S \bullet \\
\quad s.obj(sub_1) \cup .. \cup s.obj(sub_n) = \\
\qquad\qquad s.obj(sup) \qquad \text{[partition]} \\
\quad \Rightarrow \\
\quad s.obj(sub_1) \cup .. \cup s.obj(sub_{n-1}) = \\
\qquad\qquad s.obj(sup))
\end{array}
$$

which is false, as sub_n cannot be guaranteed to be empty.

9.2 Proof of the rule of promoting an association

Consider two classes A and B, where class B has subclasses $C_1 \,..\, C_n$ ($n \geq 2$). Consider also, there is an association r between A and C_1 with role ends a and b and multiplicities a_m and c_m respectively. In order to show the validity of the demotion rule, we much show that association r is also implied between A and B provided it is optional at A. The relevant parts of the proof are as follows:

$$
\begin{array}{l}
\forall\, s : S \bullet \\
\quad \mathrm{ran}(s.links(a.name)) \subseteq s.obj(C_1) \wedge \\
\quad (\forall\, j : s.obj(C_1) \bullet \\
\qquad \#\{i : s.obj(A) \mid \\
\qquad\quad (i,j) \in s.links(r)\} \in a_m)) \wedge \\
\quad s.obj(C_1) \subseteq s.obj(B) \qquad \Rightarrow \\
\quad \mathrm{ran}(s.links(a.name)) \subseteq s.obj(B) \wedge \\
\quad (\forall\, j : s.obj(B) \bullet \\
\qquad \#\{i : s.obj(A) \mid \\
\qquad\quad (i,j) \in s.links(r)\} \in a_m \cup \{0\}))
\end{array}
$$

which holds because:

- if the range of a is a subset of C_1, it is also a subset of B (as $s.obj(C_1)$ is a subset of $s.obj(B)$);

- every instance of B is associated with 0 or $x : a_m$ instances of A (as $s.obj(C_1)$ is a subset of $s.obj(B)$, and every instance of C_1 is associated with $x : a_m$ instances of A).