The Meaning of UML Models

Greg O’Keefe

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

September 2008
revised February 2010
This thesis is my own original work.

Gregory C. O'Keefe
Acknowledgements

Thanks to my supervisor Raj Goré for pushing me to do my own thing, and to clearly justify its research value. He had faith in me even when I did not. Thanks also to Shayne Flint, Clive Boughton, Lynette Johns-Boast and Stephen Mellor for many helpful exchanges on engineering and model-driven software development. I talked modelling with innumerable people at the ECMDA-FA and MoDELS conferences in 2006, and I want to thank them all even though they are too many to name. Thanks to Jean-Marc Jézéquel and his team for welcoming me to their lab between these conferences. The ANU has provided me with financial support and an exciting intellectual environment to work in. For this, I am very grateful.
Abstract

The Unified Modelling Language (UML) is intended to express complex ideas in an intuitive and easily understood way. It is important because it is widely used in software engineering and other disciplines. Although an official definition document exists, there is much debate over the precise meaning of UML models.

In response, the academic community have put forward many different proposals for formalising UML, but it is not at all obvious how to decide between them. Indeed, given that UML practitioners are inclined to reject formalisms as non-intuitive, it is not even obvious that the definition should be “formal” at all. Rather than searching for yet another formalisation of UML, our main aim is to determine what would constitute a good definition of UML.

The first chapter sets the UML definition problem in a broad context, relating it to work in logic and the philosophy of science. More specific conclusions about the nature of model driven development are reached in the beginning of Chapter 2. We then develop criteria for a definition of UML. Applying these criteria to the existing definition, we find that it is lacking in clarity. We then set out to test the precision of the definition. The test is to take an apparently inconsistent model, and determine whether it really is inconsistent according to the definition.

Many people have proposed that UML models are graphs, but few have justified this choice using the official definition of UML. We begin Chapter 3 by arguing from the official definition that UML models are graphs and that instantiation is a graph homomorphism into an interpretation functor. The official definition of UML defines the semantics against its abstract syntax, which is in turn defined by a UML model. Chapters 3 and 4 prepare for our test by resolving this apparent circularity. The result is a semantics for the metamodel fragment of the language.

In Chapter 5, we find, contrary to popular belief, that the official definition does provide sufficient semantics to classify the example model as inconsistent. Moreover, the sustained study of the semantics in Chapters 3 to 5 confirms our initial argument that the semantic domain is graphs. The Actions are the building blocks of UML’s prescriptive dynamics. We see that they can be naturally defined as graph transformation rules. Sequence diagrams are the main example of descriptive dynamics, but we find that their official semantics are broken. The “recorded history” approach should be replaced, we suggest, by a graph-oriented dynamic logic.

Chapter 6 presents our early work on dynamic logic for UML sequence diagrams and further explores the proposed semantic repairs. In Chapter 7, guided by the criteria developed in Chapter 2, we critically survey the UML formalisation literature and conclude that an existing body of graph transformation based work known as “dynamic metamodelling” is very close to what is required.

The final chapter draws together our conclusions. It proposes a category theoretic construction to merge models of the syntax and semantic domain, yielding a type graph for the graph transformation system which defines the dynamic semantics of the language. Finally, it outlines the further work required to realise a satisfactory definition of UML.
Intended Audience

The dissertation is deliberately written for workers in model driven development who do not necessarily have a strong mathematical background. Where mathematics is used, the ideas are mostly explained in plain English as well. Some of the mathematical ideas are not fully developed, but merely indicate what existing research may be relevant.
Contents

1 Logic and Modelling .......................... 1
  1.1 Logic ..................................... 2
  1.2 Conceptual Modelling ....................... 3
  1.3 Models and Statements ..................... 4
  1.4 The UML Model .......................... 8

2 The Purpose of UML and Criteria for a Definition 11
  2.1 What is a Model? .......................... 12
  2.2 Working with Ideas ......................... 19
  2.3 Diagrams and Languages ................... 20
  2.4 Semantics ................................ 24
  2.5 Metamodelling and Reflection ............. 27
  2.6 Semantic Variation Points ................. 29
  2.7 The UML Definition Evaluated ............ 30
  2.8 Conclusion ................................ 32

3 Semantic Preliminaries ....................... 34
  3.1 The Semantic Domain ...................... 35
  3.2 Object Diagrams and System States ....... 38
  3.3 Class Diagrams and Models ............... 41
  3.4 The Metamodel ........................... 50
  3.5 Reflection ............................... 54
  3.6 Reconciliation of the Two Views .......... 59
  3.7 Conclusion ................................ 64

4 Breaking the Symmetry of Association Ends 66
  4.1 Naive Symmetry ........................... 67
  4.2 Derived Property Values ................... 73
  4.3 Conclusion ................................ 77

5 Testing the Definition ....................... 79
  5.1 An Apparently Inconsistent Model ........ 80
  5.2 The Definition of UML and our Class Model 81
  5.3 UML Dynamics ............................ 87
Chapter 1

Logic and Modelling

The Unified Modelling Language (UML) is widely used in software engineering and other disciplines to express complex ideas in a way that can be easily understood. However, practical problems arise because the meaning of UML models is not well defined.

A UML model ought to be the tangible form of a shared understanding of some subject area. Discussion and negotiation are required to achieve such a shared understanding, and we should expect that models would be part of this dialog. Imagine three people working together to model some subject. Alan proposes a fragment of a model to represent certain phenomena, Betty points out an example that does not seem to be covered, Colin suggests a modification to the model that resolves the difficulty. This process can only work if the meaning of the models is clear. In current practice, arguments and confusion about the meaning of the models consume much of the available time. Apparent agreements can collapse after much has been invested in them, because the same model is understood in different ways by different people.

The aim of the present work is to find and justify a clear and precise definition of UML, which will facilitate understanding and agreement. We call this “the definition problem”. The justification part of this task is important. A myriad of proposals exist in the literature but it is not obvious what would make one a better choice than another.

Absolute precision of meaning has been achieved by formal logic. The limitation is that formal logic can only talk about abstract structures. If we wish to talk about the real world, we must assume that it resembles an abstract structure. The languages of formal logic are occasionally used in practical applications, usually when the certainty of formal deduction or semantic analysis is required. These languages are not used very widely though, because few people can understand them.

We aim to combine the precision of formal logic with the comprehensibility of UML, to achieve a language more useful than UML is in its current state. This chapter provides the background needed to understand the problem that we aim
to solve. In the first two sections, we introduce logic and the kind of modelling exemplified by UML. The third subsection relates the two fields by looking at the notion of “model” in logic, and the use of models and logic in engineering and the philosophy of science. The final section makes use of this background to explain our project from another perspective.

1.1 Logic

Although logic has a very long history going back to Aristotle around 350BC, its modern mathematical form emerged around the beginning of the 20th century with the work of Frege, Hilbert, Russell and Whitehead, and Tarski. The main motivation for this development was the foundations of mathematics. In order to eliminate questionable appeals to intuition, mathematical reasoning was to be reduced to a purely mechanical process, at least in principle. This thinking was a key influence on the development of the computer [Dav01].

Logic is the study of good reasoning. It seeks techniques for determining the validity of arguments. An argument consists of zero or more statements called premises, and a statement called the conclusion. An argument is valid if the conclusion is true in every situation where all the premises are true. Reasoning is an attempt to show that an argument is valid.

In formal logic, the statements of English or other natural languages are represented by strings of symbols called formulae. A formal language is a set of strings of a given set of symbols. Like programming languages, the languages of formal logic are usually defined by a generative grammar. Such grammars have base rules, such as “$P$ is a formula” and recursive rules such as “if $A$ and $B$ are formulae, then $A \& B$ is a formula”. When a system is defined in this way, it is possible to prove things about it using mathematical induction.

Reasoning is represented in formal logic by a purely syntactic deduction calculus, which defines a notion of proof. Each step in a formal proof is an application of one of the deductive rules of the calculus. A theory is defined to be a set of formulae which contains every formula that can be deduced from it. We can define particular theories by giving a set of axioms, just as we do informally in mathematics and science. Precisely defining a notion of proof allows us to step outside the deductive system, and prove ideas from a higher “level”. Proving properties about theories is called metamathematics. Proving properties about deductive systems as a whole is called metatheory.

The formal languages of logics are usually very simple, with as few symbols and formation rules as possible. This makes it simpler to do these metatheoretic proofs about the systems, but it does not make the formulae easier to read. Mathematics usually seeks deep consequences of simple ideas, and these simple ideas can often be expressed quite concisely using formulae of a symbolic logic. Practical problems on the other hand more often involve very complex ideas, but do not need to reason very deeply about them. Verification of computer chips is an
example of this phenomenon.

Among the most important properties a deductive system can have are soundness, completeness and decidability. A deductive system is said to be sound if every provable argument is valid, and complete if every valid argument is provable. A logic is decidable if there is an algorithm which can determine the validity of each argument. These definitions rest on the notion of validity, which in turn relies on the notion of a formula being true in a situation.

The “situations” are also formalised in mathematical logic as a structure, or interpretation. This is a collection of individuals and tells us which individual each constant represents, and what relation over the individuals each relation-symbol stands for. Given this definition, it is possible to actually define what it means for a formula to be true in a given structure. It is this mathematical definition of truth (see eg. [Hod97, §2.1]) which justifies our claim that the meaning of logical languages is absolutely precise. When a structure makes all the formulae in a theory true, we call that structure a model of the theory.

Section 6.1 will briefly introduce propositional, predicate and dynamic logics.

1.2 Conceptual Modelling

Large software projects are notoriously prone to failure, and to exceeding their budgets and timetables. Indeed “...the majority of industrial software projects are never completed but are cancelled either due to cost overruns or failure to deliver the promised functionality” [Sel01]. Despite the high cost of development, the evolution or maintenance of computer systems is often 3 or 4 times their initial development cost [Som01, §I.1.7]. The main reason for these problems is the complexity of the required systems, and this complexity continues to grow. Much of the research in software languages is intended to help manage this complexity.

The underlying concepts of programming languages have gone from pure machine notions of assembly languages to useful abstract mathematical ideas like functions and lists to more natural real world ideas like objects and messages. Techniques of abstraction and localisation have been introduced such as generalisation, encapsulation, modules and aspects. Object oriented programming languages became dominant during the 1980’s, but their origins go back another 20 years [DN66].

Research in databases has also aimed to manage complexity. In particular, the complexity of information structure. The relational model [Cod70] proved to be a useful abstraction, providing a simple but expressive half-way point between application level thinking and implementation details. Peter Chen’s 1976 paper [Che76] introduced the entity-relationship model, a natural way of describing the information of the application domain in terms of the entities that can exist and the relationships that can hold between them. Translations were defined from this model to the major implemented database models, including the relational model. Chen also provided a diagrammatic notation for this model, the entity-relationship diagram.
We examine the relational and entity-relationship models in Section 7.1. This work became the paradigm for a new research field called conceptual modelling, which also drew inspiration from work on “knowledge representation” in artificial intelligence. Conceptual modelling researchers are often housed in information systems departments in commerce faculties, whereas model driven development is more often sponsored by computer science and engineering departments. There would be benefits for all in increased collaboration between these camps.

Software engineers were also beginning to use diagrams and conceptual abstractions to capture and understand system requirements in the 1970’s. The Structured Analysis and Design Technique (SADT) was presented as a “language for communicating ideas” [Ros77]. It was widely used, and influenced the later de facto standard “data flow diagrams” [DeM78].

Object oriented visual modelling was underway by 1980 in the work of John Mylopoulos and his collaborators [Myl80]. This paper also introduces many of the ideas now found in UML: metaclasses, views, aggregation, constraints, reflection, executable models, code generation and the reification of procedure executions.

This style of modelling was popularised in the late 1980’s [SM88] and early 1990’s by software “methodologists”. Among these were the “three amigos” Grady Booch, Jim Rumbaugh and Ivar Jacobson [Boo94, RBP+91, J92]. These methodologists created UML in 1994 and 1995 by merging their different but similar visual object oriented modelling languages. Control of the language definition passed from the Rational corporation (now part of IBM) to the industry consortium the Object Management Group (OMG). The current version of UML, 2.1, is intended to support model driven software development. The OMG intends that modelling should replace programming as the primary activity of software development [MM03]. Model transformations will generate code from models just as compilers generate machine code from high level language programs.

UML has a concrete syntax consisting of 13 types of diagram and a textual constraint language. Its abstract syntax is defined using a “metamodel”, which is itself a UML model. The metamodel is very large, consisting of over 200 classes and 9 levels of generalisation. The metamodel is organised into packages which form 4 “compliance levels” using different package combination concepts. Although at first sight the diagrams seem quite intuitive, all this complexity makes it very difficult to achieve a clear understanding of the concepts underlying the language, and hence the meaning of its diagrams [FGDTS06, BG04].

1.3 Models and Statements

The astute reader may have noticed that we have already used the word “model” in at least three different ways. Analysing such ambiguities can easily result in a boring waste of ink. However, there are two main benefits we hope to gain by

---

1In fact, the UML metamodel is a MOF [Obj06a] model. MOF is a subset of the static part of UML, specialised for metamodeling.
distinguishing and relating these different notions. First, an understanding of the roles that models can play in our development processes. Second, an understanding of how we should go about making the meaning of UML models clear and precise. Section 2.1 will study notions of “model” from the model driven development literature. Here we take a larger view, drawing on logic and the philosophy of science.

UML models, like other models in science and engineering are “a representation, generally in miniature, to show the construction or serve as a copy of something” [DBB 91, The Macquarie Dictionary]. When the model is abstract rather than physical, “miniaturisation” takes the form of shedding irrelevant detail, a process sometimes called “abstraction”. We intend the models to show the construction of an intended system to programmers or model transformation tools. We intend the model to serve as a copy in various techniques of validation and verification.

Near the end of Section 1.1, we said that a structure which makes all the formulae in a logical theory true is called a “model” of that theory. We will call this a “logical model”. This seems like a purely technical definition unrelated to the dictionary sense just mentioned. If we consider the way formal logic is intended to be used, it makes more sense. Science and mathematics develop theories, which are descriptions of some structure or system. Theories in logic are intended to be formal versions of the theories of mathematics and science. Given an accurate theory in formal logic about some given system, then a logical model of the theory is a model of that system. If a logical model of the theory fails to be a model of the system, then the theory is not an accurate one.

Scientists often refer to a collection of equations or formulae as a model. Suppes [Sup60] points out that the equations are a theory, it is the logical models of that theory that should be called “model”. It is the mathematical structure that resembles the target system in some useful way, not the formulae which describe that structure. The relationship between theory, model and the system under study is shown in Figure 1.1 which is adapted from [GS06], who in turn adapted it from [Gie83]. The model description is linguistic, often a set of statements, and generally does not resemble the target system.

Another useful way of seeing the distinction is due to the philosopher Charles Peirce (1839 to 1914), and is outlined in [Als64]. Statements and names represent things as symbols which have their meaning by conventions of use, whereas models represent things as icons which resemble what they represent.

UML is a language, and its formulae are called “models”. Given a semantics
of the language, a UML “model” specifies a model which resembles the target system. That is, in the terminology of Figure 1.1, a UML “model” is actually a model description.

This would appear to be the kind of error that Suppes warns against: confusing a model description with a model. I will argue that UML is an exceptional language whose formulae can justifiably be thought of as models.

Philosophers such as Van Frassen [VF80] have suggested that the kind of resemblance required between a model and its target system is the existence of an isomorphism, or homomorphism [Mun86] from the system into the model. A considerable body of work proposes graphs and graph transformations as a theoretical foundation for UML (see Section 7.2). In such formulations, class diagrams and object diagrams are taken to be graphs, with an object diagram instantiating a class diagram just when there is a graph homomorphism

\[
\text{objectDiagram} \rightarrow \text{classDiagram}
\]

Kühne [Kuh06] proposes that the existence of a homomorphism from the system under study into the (syntactic) UML model is what enables us to use the UML model to represent the system. The same idea appears in both the philosophy of science, and the UML related work, but in the latter it is the syntax of the UML “model” which is the codomain of the homomorphism. If that really works when the details are spelled out, and if these philosophers of science are correct, then UML models, despite being syntactic, truly are models.

Readers familiar with universal algebra [BS81] should not confuse this claim with well known properties of term algebras. A term algebra is essentially the language of a given algebraic signature generated by a given set of variables. A term algebra is initial or universal in the sense that each mapping of the variables into an algebra of the same signature, extends to a unique homomorphism into that algebra. Here, the target system is acting as a model of the language, whereas for UML, we are claiming that a specific formula of the language (a UML model), acts as a model of the target system.

Even if instances of the UML abstract syntax should count as models of the target system, we also seem to think of them as defining a space of possible situations, systems or evolutions, in the same way that theories do in logic. For example, a class diagram defines a space of system snapshots. Those snapshots are themselves abstract mathematical structures, omitting the details of specific implementation technologies. Thus UML parallels logic, with the UML models playing the role of a theory, and a space of (say) abstract object oriented systems playing the role of the logical models. This parallel suggests that there may be lessons for the relatively new discipline of model driven development in the long established traditions of mathematical logic and model based science.

Peter Godfrey-Smith [GS06] distinguishes between model based and theory based science. Model based science is characterised by its efforts to understand reality indirectly by creating and studying abstract models that are known to be different from reality, but resemble it in an informative way. The deliberate inaccuracies of these models can go well beyond ignoring irrelevant detail, called “abstraction” in object-oriented and modelling literature. Frigg and Hartmann [FH06] call this
“Aristotelian idealisation”, and distinguish it from “Galilean idealisation” where deliberately false simplifying assumptions are made. Examples include frictionless planes, omniscient agents and discrete generations of population. A further kind of simplification is “approximation”, in which the description is simplified rather than the model itself. The motivating example of this is using just the first few terms of a polynomial expansion of a function. This kind of approximation will be seen in Section 7.3, where we discuss applications of logical model-checking to UML verification problems. Model checking uses languages of limited expressiveness so that the checking problem will be decidable and tractable. Since UML is much more expressive than these languages, it really is only an approximation of the UML model that is verified by such techniques.

The production of code in model driven development does not seem to be strongly model-based in this way. The models used for code generation employ Aristotelian idealisation by ignoring implementation details, but Galilean idealisation and approximation do not seem to be used. The reason is that these are prescriptive rather than descriptive. The systems modelled in science are already there, but in engineering, we make models of things that are not there, but which we would like to have. A descriptive model can describe an existing system in an inaccurate but useful way. A prescriptive model cannot be inaccurate, it simply describes the wrong system, a system which will probably not satisfy its requirements.

There is also a role for descriptive models in computer system work. Very simplified models including some false assumptions, are used for performance prediction, capacity planning and software metrics. In each case, the model resembles the target system in a specific and useful way. These exercises do not usually use UML or other model driven development languages, but they could. Model driven reverse engineering, and business process re-engineering begin by developing a descriptive model, then switch to using it prescriptively. MDD also makes use of descriptive models, for example in testing.

Although any kind of statement can be used in either way, a prescriptive statement is not very useful if we do not have control over the phenomena the statement talks about. Pamela Zave and Michael Jackson [ZJ97, GGJZ00] suggest that the vocabulary used in a system development project should be classified according to who controls the phenomenon it describes: the environment or the system. In addition, they record whether or not the phenomenon is visible to the party that does not control it. Requirements should be written using vocabulary visible to both system and environment, that is, in the system boundary. Specifications are written using vocabulary that is controlled by the system, because that is what we have power to implement. Models should only be used prescriptively when we are in control of what is modelled.

In Section 5.3 we will apply this prescriptive/descriptive distinction within UML, claiming that whilst state machines and activities can be used prescriptively, sequence diagrams ought only to be used descriptively. The actions defined in a UML activity determine what objects of the given class do, that is, they are inher-
ently prescriptive. On the other hand, the objects in an executing UML system are not controlled by sequence diagrams. This distinction will guide us when choosing appropriate ways of precisely defining the meanings of the different parts of a UML model.

We see another lesson which model-based science can offer model-driven development. Model-based science is as interested in the properties of the models as it is in reality itself, “for example, evolutionary biology has been very concerned with questions about which model systems can include the evolution of various kinds of altruism, cooperation, and reproductive restraint” [GS06]. A simple characterisation of the models which give rise to one of these properties may yield insights into how it arose in nature. It is possible that troubling phenomena in computer based systems could be mastered by a similar approach.

Mathematical logic also studies the models which satisfy each given theory. This branch of logic is known as “model theory” which “is about the classification of mathematical structures, maps and sets by means of logical formulas” [Hod97, Page vii]. One could imagine an analogous study of the classification of abstract object oriented systems by means of UML models. Insights from such a study could lead to improvements in modelling practice, and in the modelling languages themselves. The biggest obstacle to such a study is that it is currently not at all clear what these abstract object oriented systems actually are.

1.4 The UML Model

We discussed the usual dictionary usage of the word “model” in Section 1.3. With some qualification, we have seen that UML models are models in this sense. However, this sense seems different to the way it is used in the titles of two important papers discussed in Section 1.2: “The Relational Model for Large Shared Data Banks” [Cod70] and “The Entity-Relationship Model: Toward a Unified View of Data” [Che76]. The models here are general schemes for organising data, or equivalently, a set of “ontological assumptions” [Myl98] for viewing the world. The relational model assumes that the world consists of relations, the entity-relationship model assumes that the world is entities, characterised by attributes and connected by various relationships. Each artificial language takes some particular view of the world, or a notion of the kind of things or phenomena it can describe. This includes programming languages, formal logics and modelling languages such as UML. Thus, we may talk about the UML model as the same kind of thing as the relational model or the entity-relationship model, and as something quite different to any specific UML model. We will use the term “conceptual model” for things like the relational model, the entity-relationship model and the UML model.

The conceptual model of first order logic is Alfred Tarski’s notion of a structure (see eg. [Hod97, §1.1]). Its power is in its clarity, simplicity and generality. Similarly, Codd’s relational model is simple and defined in terms of elementary mathematics. On the other hand, from a modellers perspective, these conceptual
models are rather impoverished. It is hard work to squeeze every-day ideas into these restricted systems of metaconcepts.

Mylopolous [Myl98] surveys and assesses the major conceptual models. He identifies a range of “abstraction mechanisms” such as classification, parametrisation etc. used in these models. The more of these mechanisms a conceptual model incorporates, the better he rates it. Thus UML rates highly. This method seems flawed in several ways. First, it ignores the value of simplicity for deep understanding of the models. Second, it ignores the value of simplicity for modelling practice. The more mechanisms a language supports, the more difficult decisions must be made about how to model each feature of the problem space. Simplicity of the conceptual model also eases the task of writing model transformations, a key component of model driven development. However, the most serious oversight that Mylopoulos makes is assuming that a conceptual model does what it says it does.

UML involves a great many ideas or abstraction mechanisms. Each of these seems to make sense in isolation, but the interaction between them can become complicated and difficult to understand. Take for example type systems in the presence of generalisation [Bru02]. Many years of research were required before the requirements for type safety in such systems were understood, and it turned out to involve difficult concepts such as contravariant change in types. Another example is quantified modal logic [Fit99]. Both quantification and modality are well understood in isolation, but several difficult choices arise when they are combined. These difficulties are discussed briefly in Section 8.2. The number of potentially troublesome interactions in UML is huge. Our confidence is not increased by the metamodel errors (over 300 of them!) reported in [BGG04]. It seems quite possible that the proposed combination of abstraction mechanisms is simply incoherent.

A definition which turns out to be inconsistent actually defines nothing. This need not prevent apparently useful work from proceeding, as was the case in mathematics between Leibniz’ and Newton’s inconsistent formulation of the calculus in terms of infinitesimals, and Weierstrass’ clarification using limits, almost 200 years later. We should therefore not accept arguments along the lines that the definition must be OK because it is in widespread use.

The situation is even worse than we have painted so far. If all the abstraction mechanisms of UML were clearly defined, we would be ready to begin investigating whether they can be used coherently together in the way the language proposes. Unfortunately, this is not the case. As we shall see in Chapters 2 through 5, the official language definition [Obj07c] is confusing and very unclear.

In this Chapter, we have introduced formal logic and conceptual modelling. Models and statements have been distinguished and related, and we concluded that UML models are both statement and model. We have reviewed some work on the use of models in science, and compared this with model driven development. Aristotelian and Galilean idealisation and approximation are different kinds

\[\text{However, some authors see Newton’s and Leibniz’s work as vindicated by the rigorous account of infinitesimals in Robinson’s non-standard analysis [Rob96]. The point is debatable.}\]
of abstraction identified in the philosophical literature, and we saw distinct uses for these in different development tasks. We noted the distinction between description and prescription, and pointed out that prescription only makes sense if we are in control of what we want to prescribe. Finally, we noted that all artificial languages, UML included, are based on some conceptual model. Interactions between the concepts can be surprisingly complicated, which casts doubt on the coherence of large conceptual models such as that of UML.

Our aim is to solve the UML definition problem. Seen from the point of view of mathematical logic, our task is to find formal semantics for UML. Seen from the point of view of conceptual modelling, it is to find the UML model - the conceptual model underlying this modelling language. Of course the committees that designed and evolved UML have already developed a conceptual model, but as we shall see in the following chapters, it lacks the unity that the “U” in “UML” suggests. We hope that as a lone PhD student with some knowledge of logic, we can do what no committee could hope to achieve: to unify the Unified Modelling Language. This is the required first step toward a theoretical foundation for UML based model driven development. More importantly, we need this to free developers from the unproductive confusion that using UML currently entails.
Chapter 2

The Purpose of UML and Criteria for a Definition

Chapter [1] introduced our task, to solve the UML “definition problem”. Using UML does not facilitate understanding and agreement, because its definition is vague and unclear. In this Chapter, we analyse the problem and determine criteria for choosing between proposed solutions to the definition problem. We examine the purpose of UML, and the purpose of the definition of UML. The criteria we establish here will guide our work throughout the thesis, but will be especially useful when we come to evaluate the UML formalisation literature in Chapter [7].

The purpose of UML is modelling, but what is a model? In the first section we examine this fundamental question of model driven development, building on our brief discussion in Section [1.3]. We then argue that the primary purpose of UML is “working with ideas” and draw conclusions from this about what is required in a definition of the language. The following section compares UML to other languages to clarify where traditional techniques are applicable and where innovation may be required. Section [2.4] explores the reasons why we need semantics for UML, but first it is necessary to clarify the various different uses of the word “semantics”. The following two sections examine peculiarities of UML as a language, which make special demands on a semantic definition. We find that objects should be able to access the classifiers they instantiate, a property sometimes called “reflection”. We also note that the language caters for adaptation in various ways, and the definition must therefore be a flexible one. At that stage the criteria for a definition of UML are complete, and we take the opportunity to apply them to the existing official definition. Finally, a brief conclusion. The remainder of this introduction clarifies the “definition problem” we seek to solve and concludes with an overarching criterion of faithfulness to UML as it is.

The task is not to invent a new language, but to improve the definition of an existing one. The building industry has analogous situations. Sometimes a building of cultural significance is found to be structurally lacking. The builders will often suggest bulldozing it, and starting afresh, or making insensitive modifications like
replacing a timber floor with concrete.

“Underpinning” is often a more appropriate solution. Wikipedia\footnote{This source is unrefereed, indeed the present author corrected a grammatical mistake before quoting it, and could have modified it as he pleased. Since we are only making an analogy, we feel the lack of authority is not a problem.} offers the following definition:

In construction, underpinning is the process of strengthening and stabilizing the foundation of an existing building. Underpinning may be necessary for a variety of reasons:

- The original foundation is simply not strong or stable enough . . .
- The usage of the structure has changed.

\[\text{\cite{Wiki06}}\]

Too much of the UML formalisation literature takes the ham-fisted builder’s approach to the problem, largely ignoring the existing definition, omitting large parts of the language or suggesting significant changes to it. We propose instead a minimal and sensitive adaptation of the existing definition to make it strong and stable enough, and more suitable to its new usage in model driven development. Like a good restoration architect, we should carefully consider the option of leaving things as they are.

The first overarching criterion then, is that we do not fix what is not broken.

**Criterion 0.** *An improved definition of UML should not change the language or the definition any more than is needed to enable UML to fulfil its role.*

### 2.1 What is a Model?

Section 1.3 distinguished the notions of logical model and conceptual model, and related them to one another via the more general notion of model in the philosophy of science. This section focusses on the idea of model in model driven development, but again we find much needed clarification in the philosophical literature. Surveying definitions from the MDD literature, we find that the purpose of a model is to yield information about the system it models. The purpose of a particular model should correspond to the questions it is able to answer. For model driven development, the model should provide the information needed to build the modelled system, and to demonstrate that it will fulfil its requirements. One proposed definition is unsatisfactory, and we turn to Thomas Nagel’s philosophy of science to reveal the confusion. Illuminating parallels emerge between Nagel’s analysis of a scientific theory and the artifacts of model driven development. We study a mathematical account of models due to Thomas Kühne, which although it has some problems, allows us to relate model driven development to mathematical model theory and universal algebra. Results in these disciplines begin to explain...
how one system can yield information about another, and suggest the development of a strong theoretical foundation for model driven development.

Authors writing on model driven development mostly agree about what a model is. Here are two typical definitions, from Thomas Kühne and Bran Selic.

A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made. [Küh06]

The central idea [of modelling] is to produce a scaled-down version of the desired system to determine and evaluate its salient properties. [Sel01]

What are the systems salient properties? What predictions and inferences does the model allow? Bézivin and Gerbé offer a similar definition, and suggest an answer to these questions.

A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system. The answers provided by the model should be the same as those given by the system itself, on the condition that questions are within the domain defined by the general goal of the system. [BG01]

There are two distinct goals or purposes at work here, the purpose of the system and the purpose of the model. In order to allow the model to be simpler than the system itself, it must have its own goal. This will usually be to resemble the system with respect to some aspect of the systems goals. For example, one model might aim to reflect some of the systems functional goals, another its performance and reliability, and software metrics might provide a model that answers questions about its maintainability. In our view, the authors just quoted should instead have written “The answers provided by the model should be the same as those given by the system itself, on the condition that questions are within the domain defined by the general goal of the model.”

The passage clarifies how the role of a model is to be understood. For the model to answer questions in place of the actual system means that some relevant statements about the system can be reinterpreted as statements about the model. The truth-value of these statements should be constant when we switch from the model interpretation to the system interpretation. We will pursue this idea below, but first let us consider a divergent opinion about what a model is.

A model is a set of statements about some system under study. . . . we consider the model correct if all its statements are true for the [system under study]. [Sel03]

It seems that Seidewitz has made the error we discussed in Section 1.3 of confusing a model with a description. He goes on to say that the meaning of a “model” is
two-fold: what can be derived from it and how it relates to the system under study. In logical terms, this is the distinction between deduction and semantics. One could extend Seidewitz ideas by asking whether the derivation of one “model” from another is correct. In logic this is called soundness, and in formal approaches to software development, such a derivation is called a refinement.

What Seidewitz calls a “model” then, is what logicians usually called a “theory”. He is writing about models in general, but the main interest is in UML and similar modelling languages. Recall that in Section 1.3 we argued that UML is an exceptional language in that its “models” truly are models, despite also being statements. We are therefore in agreement with a slightly modified version of Seidewitz characterisation: “A UML model is a set of statements about some system under study.”

Thomas Nagel’s work on the philosophy of science identifies 3 parts to a scientific theory [Nag61]. These parts have illuminating analogies with model driven software development, and clarify the dual role of UML models as both model and description. Nagel emphasises that scientists do not explicitly present or develop their work in this way.

For the purpose of analysis, it will be useful to distinguish three components in a theory:

1. an abstract calculus that is the logical skeleton of the explanatory system, and that “implicitly defines” the basic notions of the system
2. a set of rules that in effect assign an empirical content to the abstract calculus by relating it to the concrete materials of observation and experiment, and
3. an interpretation or model for the abstract calculus, which supplies some flesh for the skeletal structure in terms of more or less familiar conceptual or visualizable materials. [Nag61, Chapter 5]

Nagel is using the word “theory” in a broader sense than the logicians sense, of a set of statements closed under deduction. Indeed this is the first of his three components, the “abstract calculus”. The basic notions of the system will be function and relation symbols in the formal language. The theory, along with the formal semantics of the language, will restrict what functions and relations these symbols can denote, thus partially implicitly defining them. This definition is merely an abstract mathematical one though, the second component of the theory connects this formal abstract structure to the observable real world. As an example of this kind of rule of interpretation, Nagel discusses Bohr’s theory that electrons have “discrete sets of permissible orbits”. The orbit of an electron is not directly observable, but the radiation that results when an electron jumps from one to another is. Each possible orbit jump is associated with a particular frequency of emitted radiation,
so that the unobservable theoretical notion is linked to an observable experimental one.

The third component of a theory is a model. In science, a model is primarily an aid to intuition, suggesting an analogy between the abstract notions of the formal theory, and more familiar ideas.

The various atomistic theories of matter illustrate the exploitation of this type of analogy. The fundamental assumptions of the kinetic theory of gases, for example, are patterned on the known laws of the motion of macroscopic elastic spheres, such as billiard balls. Similarly, part of electron theory is constructed in analogy to established laws of the behavior of electrically charged bodies. In this type of analogy, the system employed as a model is frequently a set of visualizable macroscopic objects. Indeed, when physicists speak of a model for a theory, they almost always have in mind a system of things differing chiefly in size from things that are at least approximately realizable in familiar experience, also that in consequence a model in this sense can be represented pictorially or in imagination. [Nag61, Chapter 6]

Nagel also warns that such a model will suggest things that are not strictly implied by the formal part of the theory. Let us now consider how these three components relate to model driven engineering.

One purpose of a scientific theory is to explain experimental laws. An experimental law is a statement of some observed regularity, such as that ice floats on water, or that hydrogen has certain spectral lines. A theory can employ notions that are not directly observable, like entropy, or electron orbits, but experimental laws are restricted to what can be observed. A scientific theory can explain an experimental law by deducing it. If the deduction is not a trivial one, then the proof tells us why the law is true.

There is a parallel between experimental law and scientific theory on the one hand, and user requirements and system specifications on the other. Gunter, Gunter, Jackson and Zave give a “reference model for requirements and Specifications” [GGJZ00]. User requirements state what should be observable in the system environment after the system is in place, but should not describe the internals of the system. The specification on the other hand, describes the system to be built, giving details that will not be directly seen by the user. Furthermore, the specification should entail satisfaction of the requirements. In a formal development, the requirements may be proved, using the specifications and background knowledge as premises. Thus, a theories explanation of experimental laws is parallel to an explanation of how the system specifications will satisfy the user requirements.

When creating a UML model, it is important to give textual descriptions of the intended meaning of the main model elements. For example, a class “Employee” should be accompanied by text stating, amongst other things, whether contract staff are represented by this element. Such text is “a set of rules that in effect assign[s] an empirical content” to the model.
It is not immediately clear how the third of Nagel’s scientific theory components, the model, is to be reconciled with model driven development. Different scientific theories have different models, different sets of intuitive ideas that illustrate the bare formal calculus. All UML models share a common model, in this sense, which is the UML model we discussed in Section 1.4: abstract distributed object oriented systems. Object oriented programming has been successful at least partly because it is natural to think of objects with attributes and states, interacting by passing messages. Because of its specialised semantic domain, it is not always necessary to search for further intuitive analogies to explain the system that a UML model describes. On the other hand, higher level analogies can illuminate software designs. Consider for example the names of typical design patterns [GHJV95] such as adapter, bridge, observer and visitor. We conclude that a scientific model in Nagel’s sense is analogous to a conceptual model in the sense we described in Section 1.3. A specific UML model plays the role of Nagel’s “formal calculus”.

Although Seidewitz is wrong about models in general, UML models certainly are descriptions, or “statements”, and play the role of Nagel’s “abstract calculus”. They are like a logical theory of the domain of interest, or rather how it will be after the proposed system has been introduced. Certain parts of this theory should be related to concrete observable things in the domain. UML Actors [Obj07, §16.3.1] are an obvious example. Finally, the model (as a theory) should be used to explain how the user requirements will be satisfied. The value of the model is that this can be done before the system is built.

We will return shortly to the question of how and when a model can give the same answers as the system it represents. This will require a more explicit formulation of what the model and the system are, and what the relationship between them is. The first quote in this section is due to Thomas Kühne, who develops a sophisticated account of what a model is [Küh06], which we shall use as a starting point. The “abstraction” of the system is given by a function $\alpha$ so that $M = \alpha(S)$, where $M$ is the model and $S$ is the system as a mathematical structure. One might object that the world is not a mathematical structure, and that assuming it is avoids the difficult and interesting part of the problem. We counter that, once the world is viewed according to a fixed set of clear concepts, it does take the form of a mathematical structure. Kühne decomposes the abstraction function into a projection $\pi$, other abstraction $\alpha'$ and translation $\tau$, so that $\alpha = \tau \circ \alpha' \circ \pi$. The discussion then turns to the projection and its image, the “reduced” version of the system $\pi(S) = S_r$.

With projection $\pi$ we associate any filtering of elements both reducing their number and individual information content. Projection $\pi$ is an injective homomorphism. [Küh06]

Kühne, like philosophers of science such as Mundy [Mun86], insists that the mapping between model and target system be a homomorphism, because homomorphisms “preserve structure”. However, this is not sufficient to ensure that the
map preserves or reflects truth. We will study this problem of truth preservation more carefully, but first a technical point on Kühlne’s suggestion.

Projections are not usually injective, for example \((1, 0) \neq (1, 1)\) but \(\pi_1(1, 0) = \pi_1(1, 1) = 1\). However, objects in an object oriented system have an identity independent of the values of their attributes. If the objects are considered to be tuples to which a projection can be applied, this identity would take the form of an artificial “key” field. Consider a projection \(\pi_A\), where \(A\) is a subset of the attribute names for some class. This would be injective whenever \(\text{key} \in A\). Perhaps this is what Kühlne intends. But this means that \(\pi\) cannot reduce the number of elements. He later says that “\(\pi \ldots\) creates a one-to-one relationship between target and (a subset of the) source”. This suggests that Kühlne’s \(\pi\) is a partial function, that is, a function that is only defined on part of its domain.

Since the reduced system \(\mathcal{S}_r\) was defined to be the image of this partial injective homomorphism, it has an inverse which is a total injective homomorphism \(e : \mathcal{S}_r \rightarrow \mathcal{S}\) from the reduced system into the system. Such a function is called an embedding [BS81 §2].

What can we now say about the extent to which “the answers provided by the model [are] the same as those given by the system itself”? Universal algebra [BS81] and model theory [Hod97] shed some light on this question. The relationship between the model and system is a homomorphism, and homomorphisms preserve equations. That is, any equation true in the input algebra will be true in the image of the homomorphism [BS81 Lemma 11.3, Page 79]. This does not mean that the “answer” given by the system is the same as that given by the model, because the image of the homomorphism will usually only be a small part of the system, a substructure. If the homomorphism is an embedding, this part of the system will be isomorphic to the model, that is, a copy of it.

Hodges defines preservation of truth as follows.\(^\text{2}\)

\[
\text{Let } f : A \rightarrow B \text{ be a homomorphism of } L\text{-structures and } \phi(\overline{x}) \text{ a formula of } L_{\infty \omega}. \text{ We say that } f \text{ preserves } \phi \text{ if for every sequence } \overline{a} \text{ of elements of } A,
\]

\[
A \models \phi(\overline{a}) \Rightarrow B \models \phi(f(\overline{a}))
\]

First order logic formulas are preserved by homomorphisms, so long as they have no universal quantifiers and no negations [Hod97 Theorem 2.4.3]. If the homomorphism is an embedding, then formulae with some negated atoms will also be preserved [Hod97 Theorem 2.4.1]. An embedding which preserves all first order formulae is called an elementary embedding. The image of an elementary embedding is an elementary substructure, that is, a part of the system which provides answers that are the same as those provided by complete system. The main theorem concerning elementary substructures is the Tarski-Vaught criterion, which

\(^2\)Readers not familiar with formal logic need not despair, the point is just that certain kinds of function preserve truth for certain restricted parts of the language.
Hodges warns “isn’t very useful for detecting elementary substructures in nature” [Hod97 §2.5].

We reversed the direction of Kühne’s homomorphism in order to apply some mathematics to the problem of truth preservation, but it may not have been such a good idea. A function from the model to the system can not classify the elements of the system as a function from the system to the model can. We expect objects in an object oriented system to be mapped to their classes in a class model. So we want to take answers from the image of the homomorphism and transfer them to the system which was input to that homomorphism.

Mundy [Mun86] offers a formulation of the kind of truth transfer property that we seek. His work is a philosophical study of representation, and the account that Mundy supports is known as the “structural” one. This problem is very close to our problem of when we can transfer truth from a model to a system. As Mundy puts it “the true purpose of representation is the application of the theory of the representing system to the represented system.”

The setting is a language with only relation symbols (no function symbols or constants). A homomorphism $h$ maps one structure $S$ of this type to another one $M$. (We use different symbols to Mundy to make our point in this context). Then $h$ is defined to be faithful if it satisfies the following property for every non-negative integer $n$ and every relation symbol $R$ of arity $n$ in the vocabulary.

$$\mathcal{M} \models R(h(a_1), \ldots, h(a_n)) \Rightarrow \mathcal{S} \models R(a_1, \ldots, a_n)$$

This property is dual to preservation, in that the homomorphism has jumped from the left to the right hand side. The property is also sometimes called reflection, but we will avoid that term because we use it in Section 2.5 for a different idea from object oriented programming. So each element $a$ in the target system is represented by some element $h(a)$ in the model, and whenever a simple statement is true of the representatives in the model, it will also be true of the things they represent in the real system. Note that this homomorphism would not normally be injective, many system elements would be represented by the same model element, as is the case with classes in UML.

The condition only demands faithfulness with respect to atomic statements. The situation would be complicated by the presence of logical operators, as we saw in our discussion of model theory. Also, the arrow only goes from left to right, so, if we find that some statement is false of the model, we learn nothing about the system. The more expressive a language, the more difficult it will be to determine whether this kind of truth transfer holds. However, this is the kind of condition which allows “predictions or inferences” to be made using a model. This is the kind of question we were thinking of at the end of Section 1.3 when we proposed a model theory of UML models.

---

3This is a strong sense of “representation”, as used in mathematical “representation theorems”. It should not be confused with another brand of “representation theory”, which is concerned with evaluating conceptual models by comparing them with philosophical ontologies [RRGI06, EW01].
Our analysis also clarifies the question of whether or not a model is adequate to its purpose. We could call the statements for which \( h \) is faithful the *scope* of the model. If this scope includes all the statements that are relevant to the model purpose, then the model is a good one.

To conclude this section, a model of a system is another system which, for statements relevant to the models purpose, are true of the model if and only if they are true of the system. The practical interest is that we can have information from the model that would otherwise be expensive or unavailable. We have investigated briefly how truth preservation from model to system might work. The theoretical foundations of model driven development would be very much firmer if we had a full account of how our models model.

### 2.2 Working with Ideas

Bran Selic has wisely observed that “software development consists primarily of expressing ideas” \cite{Sel03}. This is not a recent insight, the early visual language of the Structured Analysis and Design Technique (SADT) was presented as “a language for communicating ideas” \cite{Ros77}. Development involves describing the system environment and how we want it to be modified as a result of the system being introduced. These ideas must be expressed, absorbed, discussed, analysed, tested, revised and agreed on. The system itself must also be described, both at a high level, in terms of the ideas at the system environment boundary, and at the low level, using ideas about specific technologies. The high and low levels must agree, and all the ideas must be clear and free from confusion and contradiction. Indeed, the part of software development that is not about expressing ideas is mostly about generating, negotiating and translating them.

High level languages have contributed enormously to development productivity \cite{Bro87}, but ideas expressed in Fortran or Java are still far from the requirements level ideas of the human beings for whom a system is built. It is well known that requirements are not fully known, understood or agreed on at the beginning of a project, and that they will change before project completion. Hence effective software development requires the most direct possible coupling between the thoughts of the stakeholders and their expression in implementation languages.

When a skilled programmer writes code for her own purposes, this coupling is perfect. The jewels of computer programming are usually formed in this way. Consider Knuth’s TeX typesetting system, Stallman’s editor Emacs, and Tanenbaum’s Minix operating system. Extreme programming \cite{Bec99}, and other agile processes seek to couple high and low level ideas by constant face-to-face communication between stakeholders and programmers, and frequent delivery of useful code to stimulate feedback. These forms of idea coupling depend heavily on individuals. For large projects and organisations, it is desirable for the coupling of ideas to be systemic. This can be achieved by establishing model transformation and code generation chains.
The implementation platform also changes throughout the lifetime of a system. A first prototype of a proposed system is likely to employ a completely different platform to the delivered product. The delivery platform may not be known initially. The project could be tasked with deciding the platform, in which case it is desirable to conduct tests, implementing partial systems on several candidate platforms. Any successful software system will outlive the platform which implements it.

It is therefore desirable to record decisions about requirements and high-level design in a precise yet platform independent way. This provides a lasting record of the understanding achieved by the analysis of the problem. If people can read and understand the record, it enables knowledge to persist through turnover of personnel. If the record is expressed in a suitably defined language, implementations can be automatically generated from it. The record becomes a tangible and manageable form of the organisation’s “know how.”

**Criterion 1. Understandability:** UML and friends (OCL, QVT, MOF) should enable people to reach agreement on, and to directly express ideas about:

- problem domains telecommunications, finance, logistics, .
- implementation platforms Linux cluster, enterprise Java, .
- translation between these representations

The definition should enable tools to agree with people about what these expressions mean.

We agree with Steve Cook that “… for a language to be usable to drive an automated development process, it is essential for the meaning of the language to be precise” [HS05]. Without an agreed precise meaning, an automatic translators interpretation of a model might differ from that of the stakeholders. Then the delivered system might be unsatisfactory, even dangerous. The definition of UML should therefore provide a reference for those who build model translators.

In order for the language to enable people to understand and agree about their subject matter, the language must be clear and understandable. In Section 7.3 we observe that uniformity and elegance are key factors in achieving this.

### 2.3 Diagrams and Languages

UML is a language of diagrams. Diagrams do not need to conform to some defined language in order to help us communicate. People find it quite natural to express their ideas by drawing pictures, as any survey of publications, presentation slides or white-boards will verify. Most of these diagrams do not conform to any specified diagram type. If they are part of a language, it is a natural language, like English.

Natural languages are not defined, rather, linguists attempt to describe them. A description of a natural language is judged by how well it matches and predicts the
actual use of the language. A perfect description is usually not possible, because different users will use the language differently.

For artificial languages, such as programming languages and languages of symbolic logic, the situation is reversed. The description of the language is definitive, and users who do not conform to the definition are using the language incorrectly. The distinction is analogous to the one we discussed in Section 1.3, natural languages are described whilst artificial ones are prescribed. UML, although mostly diagrammatic, is an artificial language, whose correct usage is prescribed by its definition.

Sometimes the primary purpose of creating diagrams is not to communicate ideas, but rather to generate or organise them. The “mind maps” technique [Buz95] is one example. UML can be used in this way too. Building a UML model can drive the collection of information about a problem domain, and provide a convenient structure for organising that information. This role does not, however, conflict with its status as a defined language.

The mind-map book provides guidelines for creating and reading these mindmaps, which we might, very charitably, regard as a language definition. It is certainly not a precise definition, nor is it intended to be, because precision simply is not required.

Scott W. Ambler sees the role of UML in this way.

In my opinion, generative MDD is a lost cause for the current generation of developers. …I believe that modeling is a way to think issues through before you code because it lets you think at a higher abstraction level. …I typically use very simple tools, such as whiteboards and paper. … [UA03]

In a commentary attached to the amusing article “Death by UML Fever” [Bel04], Philippe Kruchten implies that UML does not need to be precisely defined.

UML is a notation that should be used in most cases simply to illustrate your design and to serve as a general road-map for the corresponding implementation.

UML can be used as documentation of code, but it is also intended as a means of specifying a system. Model Driven Architecture (MDA) [MM03] calls for complete systems to be generated automatically from UML models. If the language is not precisely defined, the generated system may not be what the model creators intended. Despite being diagrammatic, UML is, or ought to be, a precisely defined artificial language.

Some diagrammatic languages are more effective than others as instruments of communication. There is research on what makes a language effective and how to measure its effectiveness [Gur99, PC05]. Could UML be made more effective by applying some of this research? We suspect that it could, but this question lies outside the scope we have set in Criterion 0.
UML is a language of diagrams for describing systems. The diagrams are the syntax, and the systems are the semantic domain. The concrete syntax of textual artificial languages are often parsed into tree structures which are much easier to process than the linear text. These structures are called the abstract syntax. Similarly, UML has an abstract syntax which is used to transform models and generate code. UML’s subtle twist is that the class of structures used as its abstract syntax is defined using a collection of UML diagrams.\footnote{4}

Figure 2.1 shows these aspects of the language, and their relationships. On the left we have the concrete syntax, which in the case of UML are its diagrams. An example class diagram is shown inside a UML note, which we use as a kind of diagrammatic quotation to indicate that we literally mean the diagram, not what it represents or denotes. The notational conventions determine what abstract syntax is represented by a given collection of diagrams. We call an expression of UML’s abstract syntax a model. The word represents will be used exclusively for this relationship between concrete and abstract syntax, and should not be confused with semantic denotation. So the object diagram in the centre should be read as the configuration of objects represented by the quoted class diagram on the left. The right hand side shows the semantic domain, the things or situations or in this case, systems which the language talks about. When precise semantics are required, we describe this semantic domain mathematically. As we have indicated, systems which evolve over time, are often described as a set of states $\Sigma$ and a binary evolution relation $\varepsilon$ over these states.

The relation in UML between concrete diagrammatic syntax and the abstract syntax it represents, is complicated enough to be a potential source of error. Precisely defining this relationship could simplify the creation of graphical model editors, and facilitate animations \footnote{EHHS00, §6} and reverse engineering. The definition should clearly delineate concrete syntax, abstract syntax and semantics, and it should also specify the relationships between these parts. We therefore require that

**Criterion 2.** A UML definition should unambiguously define

- **concrete syntax** the diagrams and other notation
- **abstract syntax** the UML models
- **notational conventions** a unique model for each diagram collection
- **semantic domain** the abstract systems which models “talk about”
- **semantics** whether a given model is true of a given system

\footnote{4}The class diagrams of the “Abstract Syntax” sections of the definition are actually MOF diagrams, and should be interpreted according to its definition. However, the relevant part of the MOF definition is the UML Infrastructure [Obj07a], which also defines these diagrams in UML. The situation is changed considerably with UML 2.2 (issued after the thesis was written), since it gives “normative” status to an electronic version of the UML metamodel.
Figure 2.1: The five parts of a language definition
To summarise this section: we have found that diagrams need not be part of a defined artificial language, but UML diagrams are. Artificial languages do not always need to be precisely defined. However, UML should be precisely defined. Definitions of artificial languages come in two parts: syntax and semantics, but for UML the usually straight-forward notion of abstract syntax needs separate careful treatment.

2.4 Semantics

The part of the UML definition which has attracted the most research attention is its semantics, because it is very unclear. Even the word “semantics” seems to be understood differently by different people. In this section, we investigate the notion of semantics in general, and the task of defining semantics for UML in particular. We begin by comparing semantic definition with the better understood task of syntax definition.

The syntax of textual languages can be defined using a Chomsky grammar, and in the case of computer languages, this is almost always done using Backus-Naur Form (BNF). Graphical language syntaxes can be specified in a similar way, using graph grammars [BH02]. A separate grammar for the concrete syntax might not be the best way for UML though. Alternatives have been suggested which integrate the concrete syntax into the metamodel [FB05].

Grammars give a completely precise definition of a language’s syntax, leaving no doubt as to whether a given construction is part of the language or not. This precision also facilitates the construction of syntactic tools such as the parser generator YACC. Comparable techniques and tools exist for the specification of programming language semantics [Sch96], but no clear winning technique akin to BNF has emerged. English text is by far the most common form of semantic definition used. UML, although executable, is different to typical programming languages because its models should admit a wide range of possible implementation systems rather than a single specific system. That is, a UML model typically underspecifies the required system.

We must avoid the mistake pointed out in [HR04], of thinking that precisely defined language can only make highly specific statements. Although the language definition leaves no doubt about whether a semantic entity satisfies a given specification, the specification can never the less admit a wide range of possibilities. A specification such as “a truck with at least 6 wheels” can precisely define a very broad range of vehicles. Using a precisely defined language does not necessarily commit you to over-specification.

Many would argue that UML has no semantics [HR04, HS05], despite the numerous subheadings with that title in the documents which define the language [Obj06a, Obj07a, Obj07c, Obj06b]. Bran Selic [Sel04] counters these claims by collecting and summarising the scattered material on semantics from the main official document of the time [Obj05] (which was in draft form when Selic used it). He
also encourages theoreticians to study ways of making the semantics more precise.

The only real disagreement here is over the usage of the word “semantics.” This is the topic of Harel and Rumpe’s excellent article [HR04], and their position is that “semantics” is a mathematical term:

Regardless of the exposition’s degree of formality, the semantic mapping \( M : L \rightarrow S \) must be a rigorously defined function from the language’s syntax \( L \) to its semantic domain \( S \). Needless to say, an adequate semantic mapping for the full UML does not exist.

It is interesting to note these authors implicit distinction between formality and rigour. Ordinary mathematics for example is rigorous, without being formal in the logicians sense. UML is clearly informal, but Harel and Rumpe imply that this does not excuse it from rigour in defining its semantic mapping. They are not prepared to contemplate an approximate semantics, which they would probably see as analogous to being a bit pregnant. Selic on the other hand recognises the need for greater precision, but sees the existing definition as at least good enough to qualify as providing a “semantics”.

There are at least two more ways in which the word “semantics” is used. “Developers tend to use the word ‘semantics’ when they talk about the behaviour of a system they develop” [KER99]. The connection between a UML model element and the real world things it represents is also sometimes called semantics [EW01]. Both of these usages refer to a direct connection between syntax and the real world. As we discussed in Section 2.1 this representation should be seen in two steps: the model element denotes some mathematical entity in the semantic domain, and this in turn represents something in the real world. The first step is accomplished by the language definition, the second by the model authors description of her model element.

Selic is not talking about either of these things. He means basically the same thing as Harel and Rumpe when he talks about “semantics”, but like the UML definition itself, does not see the need for mathematical rigour.

It is important to note that the current description is not a completely formal specification of the language because to do so would have added significant complexity without clear benefit.

The structure of the language is nevertheless given a precise specification, which is required for tool interoperability. The detailed semantics are described using natural language, although in a precise way so they can easily be understood. Currently, the semantics are not considered essential for the development of tools; however, this will probably change in the future. [Obj07a §8]

This quote sets the scene for our investigation. It notes that some degree of precision is required to fulfil UML’s mission. It claims, with a little hesitation, that
this has been achieved without the use of rigorous mathematics. We will argue that there is a need for improvements, which we will identify.

Avoiding possible disagreements about whether or not a given system satisfies a model is enough to motivate the semantic parts of Criterion 2. Model driven development raises other questions whose answer depends on well defined semantics.

Since the abstract syntax of UML is defined by a UML metamodel, we actually require a subset of the semantics to even know whether an alleged model actually is a well-formed model. In Chapter 3 we look to the definition document to determine the semantics of this fragment of the language so that we can begin to work precisely with UML’s abstract syntax.

We need it to be clear whether or not a given model is consistent. That is, can some system satisfy this model? When we have separately modelled distinct aspects of an envisaged system, we need a system which satisfies all of the aspect models. Model consistency includes: preservation of association multiplicities and other invariants; satisfaction of pre-post-condition contracts by object behaviours; satisfaction of use-case contracts by a model; safety properties (bad things can not happen) and liveness properties (system does not get stuck). Chapter 5 will test the ability of the current definition to decide model consistency, by examining a small, apparently inconsistent model.

If a model is made more concrete as a project progresses, we may wish to determine whether the more concrete model is a refinement of the more abstract one. Indeed, we may wish to establish once and for all that a certain model transformation always produces a refinement of its input model. We tentatively call such a model transformation sound. Refinement and soundness have various mathematical definitions, but this is not the place to make these choices. Note however that it is not enough to say that one model is a refinement when it adds some detail, because we probably want to consider model transformations like the famous class to database schema example [BRST05] to be a kind of refinement. This transformation fundamentally restructures the input model rather than simply adding details to it.

The idea of transformation soundness might be of greater practical importance in model driven development than in traditional programming. High level language compilers, especially those used for production systems, are usually very mature, widely used and thoroughly tested. With model driven development, the situation is different however, partly because the execution semantics of the modelling language can be highly nondeterministic. In an MDD process, translation from model to code might be achieved by a long chain of model transformations [MM03]. Some of these might be widely used and trusted, but others might be “one-offs,” created especially for the project at hand. Automatic verification of such transformations could provide more confidence more cheaply than testing the transformations.

The ability to establish the soundness of model transformations would enable rapid development and modification of trusted systems. If the model compiler has already been shown sound, and a proof of the required properties exists for the input
model, re-certifying a modified system would only require modifying the existing proof for the changes made to the model. Just as a model is easier to modify than a low level implementation, the proof about the model would be easier to adapt than a proof about the implementation code.

We not only want these semantic questions to have definite answers, but we would also appreciate any tool support in finding these answers.

**Criterion 3.** A UML definition should settle the following questions:

- **model consistency** is there a system which satisfies all these models?
- **model refinement** is this model a refinement of that one?
- **transformation soundness** does output model always refine input?

The definition should also support maximally automatic tools to help determine the answers to these questions.

It is important to note that, although the definition of UML ought to settle the question of whether a given model transformation is sound, not all useful transformations are sound in this sense. For example, models can be used to generate tests, which are certainly not a refinement of the model from which they are derived. Soundness is primarily required for the chain of transformations which produces an implementation from the system model(s).

### 2.5 Metamodelling and Reflection

UML 2.1 is not defined in the way artificial language experts normally do business. How then is it defined? In this section and the next, we will take a brief look at the official definition of UML [Obj07c], which we will sometimes refer to as the definition.

The long and complicated story that is UML’s definition begins with another document, called the “Infrastructure Specification” [Obj07a]. This gives a UML model called the “Infrastructure Library,” which “contains all the metaclasses required to define itself” [Obj07a §7.2.8]. The Meta Object Facility (MOF) [Obj06a] builds on the infrastructure library to create a metamodelling language used to define UML in [Obj07c]. This definition of UML proper begins by including the infrastructure library.

The technique, of using a modelling language to define a modelling language is called *metamodelling*.

Metamodelling need not be circular. One can give an independent definition of the syntax and semantics of a modelling language, which we call the metamodelling language. Call the syntactic entities of this language “metamodels.” The metamodelling language semantics specify a set of instances for each metamodel. One then nominates a metamodel as the definition of the abstract syntax of a new language, which means that the abstract models of the new modelling language
are the instances of the nominated metamodel. To complete the definition of the modelling language, as we saw in Section 2.3, we would then need to specify the notational conventions by which concrete diagrams (or whatever) are interpreted into the abstract syntax, and we must provide a definition of its semantic domain and a semantic mapping from the abstract syntax into that domain.

The UML definition describes its usage of metamodelling as metacircular [Obj07a §8.1], because it uses a UML subset to define UML. Without an independent definition of the metamodelling language though, the “meta” seems like an unwarranted euphemism. In the present work, we follow a bootstrapping process which avoids the circularity. We will begin in Chapter 3 by proposing a semantic domain, and then very directly interpreting object diagrams in this domain in Section 3.2. The object diagrams will then be used to discuss pieces of abstract syntax and their semantic mappings. Finally, in Section 3.6 a second interpretation of object diagrams will be made, in which we first find the abstract syntax represented by the diagram, and then define its semantic mapping. We justify the process by showing that this second view of object diagrams is compatible with the first.

There are no dynamic elements in the fragment of UML that is used in the definition of UML, so we do not need to consider the evolution relation over system states at this stage. It is enough to clearly define when a given system state is an instance of one of these static models.

A distinction is sometimes drawn between “runtime semantics” and “repository semantics” [Obj07c §6.3]. The distinction seems to be stronger than the runtime being an extension of the repository version. This strikes us as a bad idea. In Section 2.2 we noted that uniformity is a significant aid to understanding. The main advantage of using a fragment of UML to define UML is that there is less to learn that way, but if the semantics are different at different levels, this advantage is lost. Indeed, it is more confusing than having a completely distinct language at each level. Tool and transformation reuse between metamodel and model levels also depends on this uniformity. Therefore, despite there being support in parts of the definition for distinct semantics, we shall aim for a run-time semantics which extends and includes the repository semantics.

As a result of this, the abstract syntax of the language is in its semantic domain. That is, each model is also a system state.

Having the syntax inside the semantic domain is also required in order to make sense of one of UML’s notions of instantiation. Consider a model with a class $C$ and an instance specification : $C$. Although it would be redundant in this situation, we add an $\texttt{instanceOf}$ arrow to the model, joining the instance specification to the class. Assume for now that semantic mappings take instance specifications to objects, and the classes to a sets of objects. The situation then can be depicted as shown in Figure 2.2.

Ignoring the $\texttt{instanceOfType}$ arrow for a moment, we have a neat separation between syntax on the top line, and semantics on the bottom. So we see that the $\texttt{instanceOf}$ notation in a UML diagram corresponds to “element of” ($\in$) in the system state.
The operation instanceOf, defined in MOF for the metaclass Element, “returns true if this element is an instance of the specified Class…” [Obj06a §13.3]. The arrow in Figure 2.2 marked with this name, indicates an Element, Class pair where the operation returns true. The object \( c \) is an instance of the class \( C \). The parameters of this operation include a model element and an element from the “current” system state instantiating that model. Thus the context in which this operation is called includes not only the run-time objects of the system state, but the model as well. Another example of this kind is the OCL operation oclIsTypeOf [Obj06b §7.5.9].

Systems in which objects can provide meta-data to describe themselves are sometimes called “reflective” [EN07 Chapter 1]. We have argued that UML should be reflective in this sense. However, many different usages of “reflective” are in use in the computing literature [DM95], often to do with a program’s ability to modify itself. We only use “reflective” to mean “self-describing”, or requiring self-description.

In order to define the semantics of reflective notions such as instanceOf and oclIsTypeOf, we not only require the syntax to be in the semantic domain, we actually need each syntactic model to be present in every system state which satisfies it. In Section 5.3 we will see how this works for the dynamic parts of UML.

We summarise our findings in the following criterion.

**Criterion 4.** The definition of UML should satisfy

- **unity** runtime semantics extends repository semantics

- **reflection** model contained in each of its instances

### 2.6 Semantic Variation Points

The UML definition contains a great number of “semantic variation points.” These are places where the semantics are explicitly undefined, or where a range of possibilities are allowed. Chapter 18 of [Obj07c] describes the profiles mechanism of UML, which allows subsets and extensions of UML to be defined. Model driven development may also call for domain specific languages which can interoperate with UML models. Finally, UML 2.1.2 is only the latest of many revisions of
the language, and will not be the last. For all these reasons, we require semantics which are flexible.

We need the ability to switch between the options allowed by the semantic variation points. The syntactic extensions defined in a profile will require a corresponding semantic definition, compatible with the semantics of the UML subset being extended. We will want to interpret the domain specific languages as making statements about the same class of systems that UML talks about. Let us assume, optimistically, that UML 3.0 will be published with mathematically precise yet readily understandable semantics. When UML 3.1 and UML 4.0 are introduced, we will want to know what to do to obtain models in the new language which are equivalent to our expensively produced UML 3.0 models. We may also wish to update, perhaps automatically, formal proofs associated with these models. If the semantics of the later versions are expressed in the same broad scheme, this will be much more feasible.

Criterion 5. The definition of UML must enable the language to be adapted and extended. In particular, it requires a “semantic envelope” [Sel04] which enables precise treatment of:

- semantic variation points
- profiles
- domain specific languages interoperable with UML
- later versions of UML

2.7 The UML Definition Evaluated

Having established the properties that a definition of UML ought to have, we turn now to the existing definition and ask, is it any good? In Chapter 5 we will look closely at the semantic aspects of this definition, and thus evaluate it with respect to Criterion 3. For now we take a broad, “external” view, evaluating the criteria that do not depend on the specific details.

We begin with a perfect score on Criterion 0, since no definition can be more faithful to the current definition than the current definition.

UML does not fulfil Criterion 1 so well as we could hope, because users are not currently able to easily reach agreement about the meaning of a model.

... many people are confused about what these [UML] concepts really mean and how to understand and use them [HR03].

Developers can waste considerable time resolving disputes over usage and interpretation of notation [BF98].
We have had similar experiences when attempting to extract the precise meaning of a diagram from groups of experienced UML practitioners: diverse interpretations each received vigorous support. Debate continues at the OMG over fundamental matters such as the semantics of associations and their ends [Obj, Issue #5977][Mil06]. It seems fair to conclude that there is not widespread agreement about the meaning of UML models.

In order to enable agreement, as required by Criterion 1, a language ought to be easy to understand. Implicit in many discussions of UML is the argument that the language is visual, therefore it is easy to understand. This is an empirical claim that should be verified experimentally. Nor does it follow that because UML has emerged as a consensus incorporating the most useful features from many alternatives, that it is therefore maximally understandable. This also requires empirical testing, and in fact there is some evidence against it [RBD05].

Criterion 2 can be summarised by saying that any proposed “definition” of UML should actually define it. Debates on whether or not a given diagram is a correct UML diagram, whether a system satisfies a given model and so on, should be easily resolved by referring to the definition. Indeed, if the definition was clear and understandable, these debates would seldom occur. That is to say, satisfying Criterion 1 on enabling agreement, is probably our best indication of whether Criterion 2 has been met. We must therefore conclude that the definition is also lacking in this respect.

It is not valid to infer from this that the definition lacks precision, because the lack of agreement could be the result of the definition being difficult to understand. This would be unfortunate, since it explicitly strives for understandability, even at the cost of some precision [Obj07a, §8] (quoted on Page 25). To us, it seems more plausible that the definition is neither precise nor understandable.

Turning to Criterion 3, one could hardly hope to settle questions of model consistency, refinement and transformation soundness without true definitions of the relevant concepts. It should not surprise us then that Stephen Mellor finds a lack of support for model consistency testing in the current definition [HS05]. He claims that the definition fails to detect the apparent inconsistency of his small example model. In Chapter 5 we check Mellor’s claim, and find that, given a list of charitable assumptions, the definition does in fact classify the model as inconsistent. However, this effort cost us a great deal of time and trouble, and Mellor could be forgiven for giving up earlier than we did. It is not enough to define consistency, the definition must also be usable.

Criterion 4, on reflection, is really a detail of Criterion 0, since it records what is entailed by the definition. It is hard to say whether UML’s notions of reflection really work without trying to express them in a rigorous system. Although computational reflection is a huge area of study involving many subtleties [DM95], the requirements we have given in Criterion 4 do not seem so demanding.

Criterion 5 on flexibility, is only challenging for a rigorous definition. “Semantic variation points” offer perfect flexibility. Interpretation of the metamodel diagrams by object-oriented folk-law is sufficient for current tools, which only ma-
nipulate the syntax. Without adequate support for the other criteria though, these benefits are of little use.

The current definition of UML therefore does not rate very highly against our criteria. The remainder of the present work will aim to extract the essence of the language and reformulate it clearly and precisely.

2.8 Conclusion

We have found six criteria for evaluating proposed definitions of UML. Briefly summarised, they are as follows.

0. faithful to existing UML definition
1. understandable and implementable (supports agreement and tools)
2. defines concrete syntax → abstract syntax → semantic domain
3. clarifies ideas: model consistency, model refinement, transformation soundness
4. reflective - objects can access their classes
5. flexible to support profiles and variations

The quote from the UML definition on Page 25 argues that a mathematical approach involves too much work, and is not necessary to get the job done. Whether or not Greek letters and other fancy symbols are employed, precise definitions of abstract ideas are mathematical. If we choose to ignore the accumulated wisdom of the mathematical discipline, and define things our own way, we commit the same error as “hackers” who refuse to follow established software engineering practice. Like the hackers, we are likely to get ourselves into the kind of trouble that the experts know how to avoid. One simply does not find disagreements about the meaning of definitions in mathematics, but after more than 10 years even the basics of UML are still in dispute.

Are mathematical descriptions necessarily more precise than those in English? It is possible to make precise definitions in English, after all, mathematical concepts are defined in terms of other mathematical concepts which ultimately are defined using English. Precision in English is difficult however: students asked to translate English statements into formal logic often find many distinct plausible interpretations.

To avoid these situations, mathematical definitions are almost always built using other mathematical definitions: so that the “foundation” can be as small as possible, thus reducing the risk of ambiguity. As is well known, almost all of mathematics can be defined within set theory, which uses only the logical concepts “and”, “not”, “all”, and “equals”, and the extra-logical notion of set membership: “is in”.

32
We have found that UML is a tool for working with ideas, and that the definition of UML needs to be precise and understandable. To achieve this, we must discover basic notions that can be reused and built on throughout the language. We believe that such notions are there to be found, even if they are not explicitly known by the language designers. In the following chapter, we uncover a powerful mathematical concept hidden in the UML metamodel. This will form the basis of a proposed solution to the UML definition problem.
Chapter 3

Semantic Preliminaries

The UML definition problem was introduced in Chapter 1 and we analysed it in Chapter 2. Having made it clear what we are trying to do, it is time to get on with it. This chapter begins our study of the Unified Modelling Language itself, through its current definition \[Obj07c\]. The aim is to establish enough of the semantics to work with the abstract syntax. In other words, we want to find the “repository semantics”.

In Chapter 5, we will study in detail the syntax and semantics of a tiny example UML model. We will begin with the concrete syntax: a collection of UML diagrams. This concrete syntax must be parsed into abstract syntax, because the definition describes semantics for this abstract form of model rather than giving semantics directly in terms of the diagrams. The abstract syntax that we will obtain is an instance of the UML metamodel, which is in turn given by UML class and package diagrams. Therefore, before we can obtain the abstract syntax which the example diagrams represent, we first require semantics for the fragment of UML used to write the metamodel. This repository semantics should be a subset of the full semantic definition of the language, as we determined in Chapter 2 Criterion 4.

Needing the semantics of a language before we can define its syntax might seem nonsensical, disconcerting or at least amusing to readers familiar with traditional formal languages. As we shall see in Chapter 7, many attempts to formalise UML ignore the metamodeling approach of the existing definition, and begin instead with a grammar to define the syntax. We consider this to be an undesirable and unnecessary deviation from the existing definition. Metamodelling has the advantage of allowing the reuse of tools, techniques and understanding for both modelling and language design. After all, these two tasks are essentially the same, but at different levels of generality.

This chapter therefore develops a rigorous but faithful account of most of the metamodelling required to define UML. A few items are deferred to Chapter 4. The metamodel makes heavy use of association end annotations such as \{union\} and \{subsets x\}. Chapter 4 treats the difficulties of these annotations, and also develops a proper account of generalisation and the evaluation of attributes and association
The first section of the current chapter extracts an outline of UML’s semantic domain from the current definition. We examine the core of the UML metamodel and consider its intuitive interpretation. We find deep parallels with a particular mathematical approach to graphs, and argue that UML’s metamodel, models and system states are all graphs. We then study object diagrams, which offer an ideal starting point for our semantic investigations because they show quite directly what they are intended to denote. The following section introduces class diagrams and models. We see that a system snapshot’s instantiation of a model takes the form of a graph homomorphism. This idea is well known in the literature, but we solve some difficulties and add flexibility by introducing interpretation functors. Section 3.4 applies our findings to a tiny, core fragment of the metamodel, showing that it is self-instantiating in a non-trivial way. Reflection is called for by Chapter 2’s Criterion 2 and this is addressed in Section 3.5. Here, the metamodel is included in the model, the model in the system state, and instantiation is made explicit using specially labelled graph edges. Finally, the naive view of object diagrams taken initially is replaced by a treatment consistent with the other diagrams: they are seen as representing elements of a model rather than directly denoting a system state. This change is justified by showing that the later view is a generalisation of the first.

3.1 The Semantic Domain

What are the general properties of the systems that UML models talk about? An overview is given in the definition [Obj07c §6.3], from which we have extracted the following points. A system is a set of objects, connected by links. The objects have named values attached to them, which are called attributes. Objects can act and interact and evolve over time. All the action in the system is carried out by objects, and the system evolves in discrete steps. Objects can act in parallel, and there can be several possible ways for the system to evolve. That is, systems described by UML can be nondeterministic. Objects can be created and destroyed, as can the links that join them. Objects have an identity, so we can trace an individual object’s progress in the evolution of a system containing many objects. Two objects are distinguishable even if all their attribute values are equal. Together with Leibniz law, this implies that each object has (or is) a unique object identifier.

To summarise, at any given instant, the system consists of a bunch of linked objects, which we will call a system state. A system therefore is a set of these system states, with a binary evolution relation, which tells us for each state, what can happen next.

A system state consists of objects which have named connections to one another and to other values. This kind of structure is studied in mathematics under the name “graph”. In fact a bewildering array of slightly different structures are called “graph”, so we should specify that we mean edge-labelled directed multi-
**Definition 1.2.** A graph (usually called a directed graph) consists of two classes: the class of arrows (or oriented edges) and the class of objects (usually called nodes or vertices) and two mappings from the class of arrows to the class of objects, called source and target (often also domain and codomain).

![Diagram of a graph](image)

One writes \( f: A \to B \) for ‘source \( f = A \) and target \( f = B \). A graph is said to be small if the classes of objects and arrows are sets.

**Figure 3.1:** Diagrammatic definition of a graph from [LS86]

<table>
<thead>
<tr>
<th>Property</th>
<th>classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
<td>1</td>
</tr>
<tr>
<td>type: Classifier</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 3.2:** A suggestive fragment of the UML 2 metamodel

Graphs. Each edge joins two nodes in a particular direction, and has a label. In some kinds of graph, it is not possible for two different edges to connect the same pair of nodes, which is why we call our variety a “multi-graph”. Although it is not made explicit, UML’s definition strongly suggests that system states are graphs.

Graphs are sometimes defined using a diagram, such as the one shown in Figure 3.1 taken from [LS86]. Technically, this diagram shows a small category, and a graph is defined to be a functor from this category into the category of sets. So the balloons become sets and the arrows become functions. Compare this diagram with the core of the UML metamodel, shown in Figure 3.2. The similarity between these two diagrams is the key to the following argument that UML system states, models and the metamodel are all graphs. Although it uses some technical definitions that are introduced later, it seems important to make a case for graphs before proceeding too far with them.

By adding a Labels object to the diagram of 3.1 and an arrow into it from Arrows, we have a category theoretic definition of edge-labelled multi-graphs. The resulting diagram can be seen as a category with 3 objects, or as a graph with 3 nodes. We will refer to them both as \( G \), sometimes allowing the context to determine whether it is the graph or the category we are talking about. The metamodel fragment of Figure 3.2 is clearly isomorphic to \( G \).

Although we claim that different people have different understandings of UML, the meaning of the small simple fragment used in the model of Figure 3.2 is fairly
clear. An instance of this model consists of a set of Properties and a set of Classifiers. Each Property has a name which is a string, and is associated with 2 Classifiers (possibly the same) one called its classifier the other its type. Since the multiplicities are 1 on the attribute and named association ends, these things are functions $name: Property \longrightarrow \text{String}$, and $classifier, type: Property \longrightarrow \text{Classifier}$.

But this is exactly the way the category diagram of Figure 5.1 is interpreted by a functor to yield a graph. That is, an instance of the metamodel fragment in Figure 5.2 is a functor from the diagram as a category, into the category of sets and functions. Thus, the same diagram, interpreted in the same way gives us both graphs and models. Therefore models are graphs.

However, on this view, the metamodel is a category. This is not satisfactory, because the metamodel is a model, and therefore should also be a graph. Furthermore, the models themselves should have instances. We therefore invoke a little theory to turn the metamodel as a category into another graph, and replace the functors with graph homomorphisms, which are defined in Section 3.3, Definition 3 (on Page 45). In that Section we shall see that an object diagram’s instantiation of its class diagram is just like a graph homomorphism.

A terminal object in a category is one which has a unique arrow into it from every object. There is a terminal graph, and for any graph we can find the unique homomorphism into it by a kind of inversion of its functor.

**Proposition 1.** The category of graphs has a terminal object.

**Proof.** Let $g$ be a graph. Define $!_g$ by taking each node of $g$ to the formal symbol node, each edge of $g$ to the formal symbol edge and each label of $g$ to the formal symbol label. Let $\{\text{node}\}$, $\{\text{edge}\}$ and $\{\text{label}\}$ be the nodes, edges and labels respectively of a new graph $\top$, whose source and target functions both take edge to node and whose label function takes edge to label. Any homomorphism $g \longrightarrow \top$ is equal to $!_g$. 

Like all graphs, the terminal graph $\top$ is a functor from $G$ into Set. Its peculiarity is that it takes all three of $G$’s objects to singletons. This does not convert the category $G$ into the graph $G$, because the resulting graph has only one node instead of 3. But the graph $G$ is strange in that two of its nodes “represent” an edge and a label respectively. Each of the edges “represents” an element of the source, target or label functions. Indeed, any graph could be converted into another graph in this way. The edges and labels become nodes, and the source, target and label functions become sets of edges. We call this encoding of graphs to graphs $\alpha$ (for “algebraisation”), and its inverse $\gamma$.

Note that this inverse relationship between $\alpha$ and $\gamma$ only works in one direction. If we attempt to use $\gamma$ to “decode” a graph that has not been “encoded” by $\alpha$, then it will probably have edge labels other than source, target and label, which are the only ones that $\gamma$ “understands”. That is $\gamma(\alpha(g)) = g$, but in general, $\alpha(\gamma(g)) \neq g$. However, since $G = \alpha(\top)$, we have

$$\alpha(\gamma(G)) = \alpha(\gamma(\alpha(\top))) = \alpha(\top) = G$$
These two graph manipulations play a vital role in our understanding instantiation in UML. We define $\gamma$ more carefully in the pages to come, and Proposition 2 (on Page 47) shows that $\gamma$ is a functor. We expect that $\alpha$ is also a functor. Indeed, the pair may be adjoint [Mac98 §IV], but we have not investigated this.

We have shown that there is a unique homomorphism $!_g : g \rightarrow \top = \gamma(G)$ for each graph $g$. We initially showed that the instances of the metamodel were graphs: functors from the metamodel into Set. This was unacceptable because the metamodel should be a graph, and its instances should also have instances. But $g$ is a graph if and only if there is a homomorphism $g \rightarrow \gamma(G)$.

We can therefore preserve our intuitive interpretation of the metamodel, have the metamodel be a model, and extend instantiation to system states by defining instantiation in the following way: $s$ instantiates $m$ whenever a homomorphism $s \rightarrow \gamma(m)$ exists. We will see in Section 3.3 that this notion of instantiation works for system states instantiating models, and in Section 3.4 that the metamodel instantiates itself in this way. These considerations strongly support the idea that UML system states, models and the metamodel are all graphs, and that instantiations are homomorphisms into our decoding functor $\gamma$.

We must remain cautious about our hypothesis that system states are graphs however. Firstly, the diagram of 3.2 is not quite a fragment of the UML metamodel. The type of a Property is actually a Type rather than a Classifier. However Type is an abstract class, and Classifier is its only subclass, so there are no values belonging to a Type but not to a Classifier. The other association in our diagram, classifier does not actually appear in the text of [Obj07c §7.3.44], but it does appear in Figure 7.9 [Obj07c Page 29], and is referred to in Figure 7.12 [Obj07c Page 32]. An OMG UML “issue” [Ob1 #10001] suggests that it be added, noting that “a number of the OCL expressions [in [Obj07c]] are currently written (incorrectly) with this assumption.” The second reason for caution about the graph hypothesis is that there is a lot more to the metamodel than this tiny fragment. We must give an account of all of it, not just some little bit that reminds us of some sexy mathematics. Despite these reasons for caution, we think the graph approach will work, and we will keep it as a working hypothesis.

### 3.2 Object Diagrams and System States

UML has a kind of diagram for showing objects and links, called an object diagram. An example object diagram is shown in Figure 3.3. This section introduces a somewhat naive interpretation of object diagrams that we will ultimately replace in Section 3.6. Here, we ignore the fact that object diagrams, like all UML diagrams, actually represent pieces of abstract syntax which make up a model. Instead, we see them as a notation for graphs.

In the previous section, we considered graphs as functors from a certain diagram into the category of Sets. This facilitated our argument for the parallels...
Definition 1. A directed multi-graph is a pair of sets called the nodes and the edges, and two functions: source, target : edges \rightarrow nodes. An edge-labelled directed multi-graph has an additional set, the labels, and function label : edges \rightarrow labels.

Since the only graphs we are interested in are directed multi-graphs, we will sometimes refer to these structures simply as graphs. When the context makes it clear, even edge-labelled directed multi-graphs will sometimes be called graphs or labelled graphs.

Nodes are sometimes called vertices or objects, and the edges sometimes called arrows. We will avoid the word “object” for nodes, as we already have two distinct uses for it in object oriented programming and category theory. However, some of our graph nodes will indeed be (object oriented) objects. We consider object identifiers and values to be nodes, and these nodes are connected by directed edges labelled by names. So, an object \textit{a} with an attribute \textit{grams} = 150 will be the following graph fragment.

\[
\begin{align*}
& a \xrightarrow{\text{grams}} 150 \\
& a \text{ contains } \rightarrow 150 \\
& b \text{ in } \rightarrow \text{contains}
\end{align*}
\]

Object diagrams are easy to understand, because they very directly show the state of a collection of objects and links: each object is shown as an icon containing the object’s attributes as name value pairs, and each link is shown as a line joining a pair of object icons. The ends of links can show pieces of text. By convention, these association end “names” are verb phrases which allow us to make sentences about the objects at either end. For example in Figure 3.3, we read that the apple is in the basket, and the basket contains the apple. In fact, links can have more than 2 ends, but we do not yet need to consider such links because the metamodel contains only binary associations.

What is the graph that our object diagram denotes? It will have four nodes, one for each of the objects, one for the link and one for the value 150. There will be an edge labelled \textit{grams} from object \textit{a} to 150. An arrow into \textit{a} will be labelled \textit{contains} and one into \textit{b} will be labelled \textit{in}. The sources of these two arrows is something we must investigate.

Association is a specialisation of Classifier in the UML metamodel, and links are instances of Associations. Each Association has 2 or more \textit{ends}, where it connects to a Class. Each end is described in the model by a Property. For binary
associations, each Property can belong to the Class or to the Association, giving us 3 kinds of case: both ends owned by the Classes, both ends owned by the Association, and the mixed case. For Associations with 3 or more ends, all the end Properties must be owned by the Association. Each of these possible cases are explained by the general principle we have already used for Classes and Attributes: each Property describes a set of edges, each of which comes from an instance of the Properties classifier, and goes to an instance of the Properties type.

However, we are considering the object diagram in isolation, without regard to any particular class model. Our only option then is to consider any links shown in the diagram to be ambiguous, and look separately at each possible case. In Figure 3.4 both ends are owned by the association, in Figure 3.5 one end is owned by an end Class and one by the Association, and in Figure 3.6 both Properties are owned by the end Classes. There is of course a fourth case dual to the mixed case of Figure 3.5. We will use the abbreviations CC AC and AA for these interpretations, being acronyms for Class Class, Association Class and Association Association respectively.

The mixed cases, like in Figure 3.5 can only be legitimate interpretations of class diagrams in which explicit end ownership notations occur [Obj07c §7.3.3]. This small dot at the end of the association line is an innovation of UML 2.1. Where it does not occur, either of the uniform (non-mixed) interpretations is acceptable.

This formulation of system state naturally caters for a strange possibility allowed by the UML definition. It is possible for an object to have two differently
typed values under the same name \[\text{Obj07c} \] \[\S7.3.34\], such as the example shown in Figure 3.7. Many formalisations of UML, including ours in Chapter 6, treat Properties as typed functions from objects to values. To do this in such formalisations, some trick would be required, like defining a union type. Other authors \[\text{RCA01}\] create a union type \text{Value} of all possible values, and must explicitly formalise the type restrictions on the attribute functions. In our proposal, and many graph transformation approaches, typing is achieved using a type graph. In the graph formulation, each of the two Properties would become an edge from the Class to the Property type in the dealgebraised type-graph. As a result, only properly typed values for that attribute name would map homomorphically into it. We see no need for any additional typing system.

We have not yet accounted for the class names in Figure 3.3 that is Apple and Basket. Intuitively, the class names are telling us that these objects are instances of these classes. We have decided that the model should be present in the system state, so the graphs shown in Figures 3.4, 3.5 and 3.6 are not complete system states, because the model is missing. In the next section we will find the model represented by a class diagram, which our object diagram “instantiates”. We will add that model to the system state, and connect objects \(a\) and \(b\) to their classes with special instantiation edges. There are intuitive restrictions about what can be an instance of what, and we will capture these using the idea of a graph homomorphism. In the following section we will continue up the metamodelling hierarchy, adding a self-instantiating metamodel.

We now have a tentative definition of system state, and a preliminary, temporary semantic mapping for object diagrams. Next we begin to take seriously the idea that diagrams represent models, and thus study the notational conventions (Figure 2.1 on Page 23) for parsing class diagrams. Since models are system states are graphs, and object diagrams are now established as a notation for graphs, we can use object diagrams to display our models.

### 3.3 Class Diagrams and Models

Now we are ready to consider the most basic kind of class diagrams, and we begin to use the notion of a model. We have already mentioned some (meta)classes from the metamodel: Association, Property and Class. The model will be a system state in which the (meta)objects belong to (meta)classes such as these. We will use
Apple
grams: natural
contains
Basket

Figure 3.8: A class diagram

object diagrams to show metamodel elements, as is the common practice [Obj07c, Figures 11.21, 14.23] [BG04, Mil06]. The UML definition [Obj07c, §7.4] tells us what metamodel elements are represented by the parts of the class diagram. For now, we will not attempt to determine all the details of these metamodel elements. "Navigating the metamuddle" [FGDTS06] is a complicated and difficult task which we leave to Chapter 5. Here, we simply wish to clarify for a small, static fragment of UML, what is a model and what does it mean for a system state to be an instance of it.

Figure 3.8 shows a class diagram which intuition says should be instantiated by the system state we discussed in the previous section (Figure 3.3 on Page 39). In this section we will parse the diagram to obtain its abstract syntax, and define what its instances are. In the next section we will show that it is an instance of the metamodel, and that the metamodel is an instance of itself.

In the previous section we mentioned the new UML notation for ownership of association end Properties. As we noted in the previous section, Associations with 3 or more ends own all their end Properties [Obj07c, §7.3.3, constraint 5], so the CC (CCC, C*) reading is not an option. Our class diagram does not make use of this “dot,” so we are left with two legitimate interpretations of the class diagram. Either both association ends are owned by the Association (AA) or both are owned by their classes (CC). For object diagrams, this choice is determined by the ownership of Properties in the model, but it is our interpretation of class diagrams that determines this Property ownership, so we have nothing on which to base this choice. Since we will see an example of a CC interpretation of a class diagram below, we take this opportunity to show an AA reading.

Figure 3.9 shows the main metamodel elements of the model represented by the AA reading of the class diagram in Figure 3.8. The CC reading is similar, but the classifier ends at the Association would go to Classes instead. This would appear to leave the Association isolated, but in fact there are other links not shown in our fragment, including two with ends association and memberEnd which would connect the Association and its end Properties.

We have found an object diagram for the elements of our model, but as we mentioned in the previous Section, these diagrams are ambiguous, and depend on the model they instantiate to resolve this ambiguity. In our case, the model being instantiated is the UML metamodel. This is given only as class diagrams without end ownership notations, and are thus also ambiguous. That is, we do not know whether the associations shown in the UML metamodel class diagrams own their
end Properties, or whether these Properties are owned by the end Classes. There is an electronic form of the metamodel, issued by the OMG in the proprietary format of IBM’s Rational Rose tool. This would give an answer to the question, but not an authoritative one, since it lacks “normative” status.\(^1\)

The choice then seems to be ours to make, which is good, because the account of instantiation as graph homomorphism that we wish to develop works much more simply if we take a CC reading of the metamodel. This means we can very directly see the object diagram of Figure 3.9 as a labelled directed graph. The link nodes will be isolated, and each link shown in the object diagram will denote a pair of edges between the object nodes at its ends, in opposite directions. One of each pair will have an arbitrary label, determined by our interpretation of the metamodel diagrams. We will adopt the convention that the partner of an edge labelled \(x\) will be labelled \(x^{\text{Inv}}\), so for example the link between \(\text{grams} \) and \(\text{natural} \) in Figure 3.9 will denote a pair of edges \(\text{grams} \xrightarrow{\text{type}} \text{natural} \) and \(\text{natural} \xrightarrow{\text{typeInv}} \text{grams} \).

An important difference between this model and the system state that we feel should instantiate it, is that the Properties are nodes in the model, but edges in the system state. The definition seems to support this. “Property represents a declared state of one or more instances in terms of a named relationship to a value or values” [Obj07c, §7.3.44, Description] (emphasis added). Graph treatments of UML-style modelling languages usually define instantiation as graph homomorphism, which

---

\(^1\) Updated versions of these files, in the standard .xmi format became normative with UML 2.2 [Obj09] in February 2009. These files confirm that the metamodel classes own their association ends, i.e., the CC reading is applied to the metamodel diagrams.
seems to capture the intuition well (eg [BH02], graph homomorphism is defined below). This means that the labelled edges that represent an objects properties must map to similarly labelled edges in the type graph. Can we reconcile Properties as nodes with Properties as edges?

Another way of seeing Figure 3.9 is as the “algebraic” form of a new graph. Recall that a graph is a set of a set of edges with source and target functions into a set of nodes, and a label function into a set of labels. The Properties in the graph of Figure 3.9 each have unique outgoing edges labelled type, classifier and name. Thus the nodes that these arrows originate from become the edges of our new graph. The nodes of the new graph are the destination nodes of the classifier and type edges, and the labels the destination nodes of the name nodes. The resulting graph is shown in the traditional graphical way in Figure 3.10.

The similarity between the graphs of Figures 3.4 (Page 40) and 3.10 is obvious. Indeed, there is an obvious label preserving graph homomorphism which sends each object to the class that it should instantiate. We therefore propose to “interpret” the class model using “dealgebraisation”, to obtain a type graph which determines the models instances.

Dealgebraisation turns out to be a functor, an idea from category theory. We discuss the connections with category theory a little more after defining the graph part of the interpretation functor in Definition 2. We then show it is a functor in Proposition 2. The functor will be replaced in Definition 2 by a minor technical variant which we will call \( \gamma \). The interim version we are about to introduce is therefore called \( \gamma' \). A couple of notational conventions will allow us to be more concise. When the context makes it clear which graph is being discussed, we will write \( e : x \to y \) to abbreviate \( \text{source}(e) = x \land \text{label}(e) = \ell \land \text{target}(e) = y \), and \( x \ell y \) for \( (\exists e \in \text{edges})e : x \to y \).

**Definition 2.** Let \( g = (\text{nodes}, \text{edges}, \text{labels}, \text{source}, \text{target}, \text{label}) \) be a labelled graph. Let \( \text{edges}' \) be the nodes \( n \in \text{nodes} \) where \( n \) has exactly one outgoing edge labelled “classifier”, exactly one labelled “type”, and exactly one labelled “name”\( b \). Let

\[
\begin{align*}
\text{source}' &= \{(e', n') \mid e' \in \text{edges}' \land e' \overset{\text{classifier}}{\to} n'\} \\
\text{target}' &= \{(e', n') \mid e' \in \text{edges}' \land e' \overset{\text{type}}{\to} n'\} \\
\text{label}' &= \{(e', \ell') \mid e' \in \text{edges}' \land e' \overset{\text{name}}{\to} \ell'\}.
\end{align*}
\]
We complete the definition of a new labelled graph

\[ \gamma'(g) = (\text{nodes}', \text{edges}', \text{labels}', \text{source}', \text{target}', \text{label}') \]

by setting

\[ \text{nodes}' = \text{Range(source')} \cup \text{Range(target')} \] and
\[ \text{labels}' = \text{Range(label')} \]

We have informally used the idea of graph homomorphism, which is a structure preserving map from one graph into another. The idea is defined as follows.

**Definition 3.** Let

\[ g = (\text{nodes}, \text{edges}, \text{source}, \text{target}) \]

and

\[ g' = (\text{nodes}', \text{edges}', \text{source}', \text{target}') \]

be graphs. A graph homomorphism \( g \rightarrow g' \) consists of a pair of functions \( n : \text{nodes} \rightarrow \text{nodes}' \), \( e : \text{edges} \rightarrow \text{edges}' \) such that

\[ n(\text{source}(x)) = \text{source}'(e(x)) \]
\[ n(\text{target}(x)) = \text{target}'(e(x)) \]

for every \( x \in \text{edges} \).

As usual, if both components of a graph homomorphism are bijective (one to one onto) we call it a graph isomorphism. In this case, its inverse is also a graph homomorphism.

Category theory is used heavily in the literature on graphs and graph transformations [EEP10]. The work presented in this dissertation does not go deep enough to need this kind of machinery, but we relate it to the category theoretic setting where we can, in the hope that it will facilitate future work. Readers unfamiliar with category theory will not miss anything substantial. The standard work on category theory is [Mac98], and a gentle introduction for computer science is [Pie91].

A category is like an (unlabelled directed) graph, except that the edges (called arrows) can be composed to form new arrows. Each node (called an object) has a special identity arrow, which when composed with another arrow, yields that arrow unchanged. One example of a category has graphs as objects and graph homomorphisms as arrows. Category, as a kind of mathematical structure, has its own notion of homomorphism: a homomorphism between categories is called a functor. There is also a notion of natural transformation between two parallel functors, that is, functors between the same pair of categories.

Graphs themselves may be defined as functors from the category \( \bullet \rightarrow \bullet \) to the category of sets. When graphs are seen this way, graph homomorphisms are natural transformations [Mac98 §I.4] between them.
Definition 3 does not mention graph labels. The following definition extends it to include a mapping of labels. Our apple in a basket example of homomorphism as instantiation motivates a stronger notion of labelled graph homomorphism. In that example, edges only map to edges with the same label.

**Definition 4.** Let
\[
g = (\text{nodes}, \text{edges}, \text{labels}, \text{source}, \text{target}, \text{label})
\]
and
\[
g' = (\text{nodes}', \text{edges}', \text{labels}', \text{source}', \text{target}', \text{label}')
\]
be labelled graphs. A labelled graph homomorphism is three functions \( n : \text{nodes} \rightarrow \text{nodes}' \), \( e : \text{edges} \rightarrow \text{edges}' \) and \( l : \text{labels} \rightarrow \text{labels}' \) such that \((n, e)\) is a graph homomorphism, and
\[
l(\text{label}(x)) = \text{label}(e(x))
\]
If \( l = \text{id}_{\text{labels}} \), (and therefore \( \text{labels} \subseteq \text{labels}' \)) we say \((n, e, l)\) is label-preserving.

The inverse of a label preserving graph isomorphism is also a label preserving graph isomorphism.

We can now name some categories of graphs. The composite of two (label preserving) homomorphisms is also a (label preserving) homomorphism, and the identify function over a labelled graph is a (label preserving) homomorphism. We therefore have two categories whose objects are labelled graphs. The one with all homomorphisms will be called \( \text{Gr} \), the subcategory whose arrows are label preserving graph homomorphisms we call \( \text{LPG} \).

Definition 2 gives us a function from graphs to graphs. This will serve as the object part of a functor, but a functor \( F \) must also take each arrow \( f : g \rightarrow h \) to an arrow \( F(f) : F(g) \rightarrow F(h) \), and it must preserve the categories structure: its composition and identity arrows [[Mac98, §1.3][Pie91, §2.1]]. Since the edges and nodes of \( \gamma'(g) \) are all nodes of \( g \), we can obtain suitable arrow and node components for \( \gamma'(f) \) by restricting the node component of \( g \). We write \( f|_A \) for the function \( f \) restricted to the set \( A \).

This will only work if the nodes in \( g \) that represent edges are mapped by \( f \) to nodes in \( h \) that also represent edges. The labels type, classifier and name play a special role when we use a graph to represent another graph in algebraic form. The edges of the original graph become nodes with exactly one outgoing edge for each of these labels, and this is how \( \gamma' \) (Definition 3) recognises them. Any node which has these edges can only be mapped by a label preserving homomorphism to a node that also has them. However, there is no guarantee that this latter node has only one outgoing edge with each of these three labels. It might for example have two outgoing edges labelled type, and thus not be turned into an edge by \( \gamma' \). Rather than restricting the homomorphisms to account for this, it seems more natural to restrict the graphs to those that seem to make sense as algebraic representations of other
graphs. Some nodes will need to have type edges but no classifier for example, but we want to exclude nodes that have all three, but have more than one edge with the same label.

**Definition 5.** A graph node is tcn if it has exactly one outgoing node with each of the labels type, classifier and name. A graph is tcn if each node with outgoing type, classifier and name edges is tcn. We write $\text{Gr}_{\text{tcn}}$ and $\text{LPG}_{\text{tcn}}$ for the subcategories of $\text{Gr}$ and $\text{LPG}$ whose objects are tcn graphs.

**Proposition 2.** The function $\gamma' : \text{Graph} \to \text{Graph}$ of Definition 2 is the object component of a functor $\text{LPG}_{\text{tcn}} \to \text{Gr}_{\text{tcn}}$.

Proof. Let $g$ and $h$ be tcn graphs and $f : g \to h$ a label preserving homomorphism. Define the three components of $\gamma'(f)$ as follows.

$$
\begin{align*}
\text{edges}(\gamma'(f)) &= \text{nodes}(f)|_{\text{edges}(\gamma'(g))} \\
\text{nodes}(\gamma'(f)) &= \text{nodes}(f)|_{\text{nodes}(\gamma'(g))} \\
\text{labels}(\gamma'(f)) &= \text{nodes}(f)|_{\text{labels}(\gamma'(g))}
\end{align*}
$$

If $e$ is an edge $e : t \to c$ in $\gamma'(g)$ then it is a node in $g$ with outgoing edges $e \xrightarrow{\text{type}} t$, $e \xrightarrow{\text{classifier}} c$ and $e \xrightarrow{\text{name}} n$. These are the only outgoing edges from $e$ with these labels. The three edges are mapped by $f$ to edges $\gamma'(f)(e)$.

Therefore $f(e)$ is an edge in $\gamma'(h)$. More specifically, $f(e) : f(t) \xrightarrow{\text{type}} f(c)$. But $f(e) = \gamma'(f)(e)$, so we have shown that the edge component of $\gamma'(f)$ takes edges to edges. Each node in $\gamma'(g)$ is analogous to the nodes $t$ and $c$, being at the far end of a type or classifier edge out of a tcn node. Since $f$ preserves tcn, $f(t)$ and $f(c)$ will be nodes in $\gamma'(h)$. But $\gamma'(f)(x) = f(x)$ for $\gamma'(g)$ nodes $x$, so, $\gamma'(f)$ takes nodes to nodes. Similarly, the labels of $\gamma'(g)$ map to labels of $\gamma'(h)$, because the tcn property and the label name preserved by $f$. We have now shown that $\gamma'$ takes label preserving homomorphisms to homomorphisms. It remains to show that it respects composition and identity.

Let $k$ also be a tcn graph, and $j : k \to g$ a label preserving homomorphism. We now show that $\gamma'(f \circ j) = \gamma'(f) \circ \gamma'(j)$. We have

$$
\begin{align*}
\text{edges}(\gamma'(f \circ j)) &= \text{nodes}(f \circ j)|_{\text{edges}(\gamma'(k))} \\
&= \text{nodes}(f|_{\text{edges}(\gamma'(g))} \circ j|_{\text{edges}(\gamma'(k))}) \\
&= \text{nodes}(f|_{\text{edges}(\gamma'(g))}) \circ \text{nodes}(j|_{\text{edges}(\gamma'(k))})
\end{align*}
$$

47
since the node component of \( j \) takes \( \gamma'(k) \) edges to \( \gamma'(g) \) edges. Similar reasoning applies to the node and label components of \( \gamma'(f \circ j) \).

Finally, each component of \( \gamma'(id_g) \) is a subset of the node component of \( id_g \), and thus the identity function over the nodes, edges and labels of \( \gamma'(g) \) respectively. We therefore have \( \gamma'(id_g) = id_{\gamma'(g)} \).

In what follows, we will find uses for graph interpretation functions other than our dealgebraisation function \( \gamma' \), and so we make the following definition generic in this respect.

**Definition 6.** We say that a labelled graph \( s \) is an \( f \)-instance of a labelled graph \( m \) iff there is a label preserving graph homomorphism \( h : s \rightarrow f(m) \).

This is a parametric semantics for graphs. Given an interpretation function \( f \), we have a set of instances for each graph. The expression \( s \rightarrow f(m) \) is a rather degenerate example of a style of diagram from category theory, called a commuting diagram. Different parts of UML models are interpreted in different ways. Later in this chapter, and more fully in Chapter 8, we will achieve this by using several interpretation functions in a commuting diagram. This kind of modular flexibility is a good thing, because we aim to cater for UML’s “semantic variation points”.

We noted before that there is a label preserving homomorphism between our example system state and the interpreted model. With our definitions, this justifies the following statement of instantiation.

**Proposition 3.** The graph of Figure 3.4 (Page 40) is a \( \gamma' \)-instance of the graph of Figure 3.10.

Thus as we would hope, the denotation of the object diagram in Figure 3.3 (Page 39) is an instance of the denotation of the class diagram in Figure 3.8 (Page 39).

The reader might wonder what is the point encoding a graph in algebraic form, as another graph. Why not just store the model in its dealgebraised form, and call graph homomorphisms instantiations? This would be sufficient to define the semantics of models by determining what its instances are. However, the model syntax must exist in a form that can be navigated by modelling and code generation tools. We see the nodes of our graphs as data values, and the edges as the means of navigation through these values. For tools to process models, all the model elements must be accessible. This includes the Properties which dealgebraisation turns into edges. Access must be via edge labels that are known in advance, which in the algebraic form, are the association end names given in the metamodel.

The direct, instantiation-as-homomorphism view would also fail to account for metamodels which are self-instantiating. Under that view, every model would be self-instantiating, since the identity function over a graph is certainly a label preserving graph homomorphism. We will see in the next Section that the definition we have given does provide a non-trivial account of self-instantiation.
There would probably be uses for a weaker notion of instantiation, using labelled graph homomorphism rather than label preserving ones. This would allow for synonyms that may arise, for example as a result of different “viewpoints” being modelled by separate teams.

We come now to a point about UML that many find surprising. Objects can instantiate more than one class. The first way this can happen is through generalisation. “Where a generalization relates a specific classifier to a general classifier, each instance of the specific classifier is also an instance of the general classifier.” [Obj07c §7.3.20].

UML also allows an object to instantiate several different classes simultaneously, regardless of their relationship in the specialisation hierarchy. The description of InstanceSpecification and the reclassifyObjectAction makes this clear [Obj07c §7.3.22, §11.3.39]. The definition’s explanation of GeneralizationSet also describes cases of objects instantiating two or more classes. Several Generalizations can be grouped using a GeneralizationSet [Obj07c §7.3.21], which can be marked \{overlapping\}. This mark in the diagram indicates that the GeneralizationSet attribute isDisjoint has the value false, which is actually the default. The definition gives an example. “Person could have two Generalization relationships involving two specific (and non-covering) Classifiers: Sales Person and Manager. This GeneralizationSet would not be disjoint because there are instances of Person that can be a Sales Person and a Manager” [Obj07c §7.3.21]. This example makes it clear that one object can be an instance of two different classes, neither of which specialises the other.

The account of instantiation we have developed so far only allows an object to instantiate a single Classifier. What we need is a type graph whose nodes are actually sets of nodes. Thus we define a power-graph construction that takes a graph \( g \) and yields a new graph \( \mathcal{P}(g) \) (or \( 2^g \)) which has sets of \( g \) nodes as its nodes. For the edges, we put an edge from one set of nodes to another if it goes from a member of one to a member of the other in the input graph. That is, if \( a \xrightarrow{\ell} b \) is in \( g \) and \( a \in A \) and \( b \in B \), then \( A \xrightarrow{\ell} B \) is in \( \mathcal{P}(g) \). The definition below achieves this by defining for each arrow \( x \), two sets of sets of nodes, sources\( (x) \) which is all the sets of nodes that contain \( x \)’s source, and similarly for targets\( (x) \).

**Definition 7.** Given a graph \( g \) we define the power-graph \( \mathcal{P}(g) \) of \( g \). Set nodes\( (\mathcal{P}(g)) = \mathcal{P}(\text{nodes}(g)) \). For \( x \in \text{edges}(g) \), let

\[
\begin{align*}
\text{sources}(x) &= \{ \{ \text{source}(x) \} \cup S \mid S \subseteq \text{nodes}(g) \setminus \{ \text{source}(x) \} \} \\
\text{targets}(x) &= \{ \{ \text{target}(x) \} \cup S \mid S \subseteq \text{nodes}(g) \setminus \{ \text{target}(x) \} \}
\end{align*}
\]

and

\[
\text{edges}(\mathcal{P}(g)) = \bigcup_{x \in \text{edges}(g)} \text{sources}(x) \times \{ x \} \times \text{targets}(x).
\]

Then we set \( \text{source}((s,x,t)) = s \) and \( \text{target}((s,x,t)) = t \) and \( \text{label}((s,x,t)) = \text{label}(x) \) for each \((s,x,t)\) in \( \text{edges}(\mathcal{P}(g)) \).
Now, if \( s \) is a \((P \circ \gamma')\)-instance of \( m \) we have a label preserving homomorphism \( h \) whose node component takes each node of \( s \) to a set of nodes of \( m \). This allows objects to have properties from several different classes. In the Section 3.5 system states will contain their models, and instantiation will be given explicitly by special edges. We will use power-graphs to check that these explicit instantiations are correct, in the sense that they can be extended with an edge component to yield a homomorphism. If we take the power-graph of a model, and map each object in a system state to a set of nodes in the model, then this map is extendible to a label preserving homomorphism iff all the edges match an edge between some pair of nodes that the ends of the edge get mapped to.

This is however, a very weak notion of instantiation. For example \( P(g) \) contains a node which is the set of all nodes from \( g \), and has a copy of every edge of \( g \) as a reflexive edge. Thus any graph whose edge labels are among the edge labels of \( g \) can be mapped homomorphically into \( P(g) \) by sending all the nodes to this node. This is effectively saying that every object and value is an instance of every Classifier.

To deal with generalisation, we require that “each instance of the specific classifier is also an instance of the general classifier.” [Obj07c §7.3.20]. That is, we do not want anything mapped to a set of nodes that contains some special class, but does not contain the class it specialises. The Generalizati ons in a model partially order its classes. Correct semantics for generalisation can be obtained by taking the subgraph of \( P(g) \) whose nodes are closed upwards under this partial order.

Models, instantiation and the semantic mapping for very simple class models now seem to be explained by our provisional formalisation. Although there are four levels to the UML modelling hierarchy, we will be content to extend our account to a third level, where this third, top level is self-instantiating.

### 3.4 The Metamodel

In this section, we make the tiny core metamodel fragment of Figure 3.2 (on Page 36) explicit as a graph, which we shall call \( mm \). We find some ways of simplifying it even further, show that it is non-trivially self-instantiating and that the model from the previous section is an instance of it.

To find the model represented by the metamodel class diagram fragment of Figure 3.2 we will follow the same procedure as in the previous Section, except this time we assume that the association end Properties are owned by their classes (the CC reading). The resulting object diagram showing the elements of the metamodel is shown in Figure 3.11.

Because our fragment omits all but two associations from the real metamodel, and because we have chosen the CC reading, the two Association metaobjects are not linked to anything. This means that the instances of this model will have isolated links, as we discussed before. These isolated links are of no use in determining the semantics of the models, since \( \gamma' \) does not look at them. Neither are they
Figure 3.11: Key metamodel elements of the Metamodel fragment of Figure 3.2
useful for tool navigation of the models. The $xInv$ Properties will result in corresponding edges in $\gamma'(mm)$ and thus allow inverse edges in the models. Although navigation using these edges would be useful in modelling tools, they are not illuminating for our present purposes since they are just backwards copies of the edges we are interested in.

We therefore introduce an economised version of the CC reading of class and object diagrams. The CClite reading of a class diagram will be the object diagram given by the CC reading, except without the Associations and without Properties for anonymous association ends, and obviously without the outgoing classifier and type links of these Properties. The CClite reading of an object diagram will be the graph given by the CC reading, except without link nodes, and without edges for anonymous link ends. Therefore, the CClite reading of an object diagram $OD$ will instantiate the CClite reading of a class diagram $CD$ iff the CC reading of $OD$ instantiates the CC reading of $CD$. When we draw the graph obtained in this way from an object diagram, the graph will appear as a minor notational variation of the object diagram. Thus we can work with less cluttered graphs, whilst retaining equivalent instantiation semantics, as we shall see below.

Note that this is only a temporary convenience, applicable here because of the limited static semantics that we are interested in. In a legitimate UML system state, each link belonging to an $n$-ary Association is one node and $n$ edges. When we extend our definitions to cater for system dynamics, this will be stated formally because semantics for Actions and navigation expressions will depend on it.

The CClite reading of the metamodel fragment will be the “inner” part of Figure 3.11 it will exclude the two Associations and the classifierInv and typeInv Properties. We obtain the model denoted by this object diagram using the CClite interpretation. This is not because it is the metamodel, but because it is an instance of the metamodel, which we have decided has association end Properties owned by its Classes.

As a further economy, the Property nodes are identified with their String names. The resulting metamodel is shown as a graph in Figure 3.12 Its $\gamma'$ interpretation is shown in Figure 3.13 This is the same as the interpretation of the larger version with distinct Properties and Strings, so it has the same $\gamma'$-instances or static semantics.

If we were to define edge-labelled graphs as functors, in the way Lambek and Scott did with the diagram we reproduced in Figure 3.1, then Figure 3.13 is what we would use as the source category, except with different names. It also has a comforting resemblance to the class diagram from which it originated, in Figure 3.2 on Page 36.

To make sense of the dual roles of the Property/String nodes, we will employ the power-graph treatment of multiple instantiation, introduced in the previous section. There are many label preserving homomorphisms $mm \rightarrow P(\gamma'(mm))$, but we show the image of the one which captures our intended object instantiation in Figure 3.14 Since it is a homomorphic image, it is a subgraph of $P(\gamma'(mm))$.

Now let us consider the apple-in-a-basket model again. Its metamodel elements
Figure 3.12: The metamodel fragment as a graph

Figure 3.13: $\gamma'$-interpretation of the metamodel fragment.

Figure 3.14: Subgraph of the metamodel power-graph given by the self-instantiation homomorphism.
object diagram is shown in Figure 3.9 (Page 43). Like the object diagram we just
looked at, it denotes an instance of the metamodel, and thus we should use the
CClite reading to obtain a graph from it. The fact that the object diagram showing
its model elements was obtained by an AA reading of a class diagram does not
affect this. If we were reading an object diagram intended to instantiate that apple-
in-a-basket model, then we would use the AA reading, but this is an instance of the
metamodel, which we read as CClite.

It is not hard to see that there is a label preserving homomorphism from the
CClite reading of Figure 3.9 into the graph of Figure 3.13. We map the Properties
into Property, the Classes and Associations and DataType into Classifier. For now
we think of these specialisations of Classifier to be merely synonyms for it. This
naive view will be refined in Chapter 4. Each of the strings will map to String, and
then each edge has a suitable image. Thus the apple-in-a-basket model truly is a
\( \gamma' \)-instance of the metamodel. Since there is an obvious injective homomorphism
\( \gamma'(mm) \longrightarrow \mathcal{P}(\gamma'(mm)) \) which sends each node to the singleton set containing it,
the model is also a \( \mathcal{P} \circ \gamma' \)-instance.

Similar reasoning applies to the metamodel \( mm \) itself. Looking at the object
diagram of Figure 3.11 with the now familiar CClite reading, and the graph \( \gamma'(mm) \)
in Figure 3.13, it can be seen that we have a label preserving graph homomorphism
\( mm \longrightarrow \gamma'(mm) \), and thus the metamodel is a \( \gamma' \)-instance of itself. This applies
whether we consider the full object diagram of Figure 3.11 or just the subset of it
given by the CClite reading of the metamodel fragment class diagram.

3.5 Reflection

Instantiation as we have treated it thus far is really only implicit. One graph instan-
tiates another if a certain homomorphism exists. However, in implemented object
oriented systems, there is some specific collection of classes, and an explicit instan-
tiation relation assigning each object to one or more of these classes. For example
in an object oriented program, each object will have been created by a constructor
method of one of the programs classes. A model repository is another example. It
implements some metamodel, and this implementation sees each of the elements
in the repository as an instance of a specific element of that metamodel, so that
for example, one can iterate through all the classes in the model to generate code.
We also still need to account for headings of the form \( \textit{someObject} : \textit{SomeClass} \)
which occur in object icons. Our Criterion 4 (on Page 29) requires that objects can
access the classes they instantiate and thus enjoy a kind of self-knowledge. Also,
the classes should be able to access their metaclasses and so on. The purpose of
this section is to develop the required explicit instantiation.

The most obvious way to achieve all this is to take the union of the system
state, the model and the metamodel (and any further levels in the hierarchy) and
connect the relevant pairs of nodes with edges labelled \( \textit{instanceOf} \). This resembles
Definition 8. If every node of a graph \( g \) has an outgoing `instanceOf` edge, we say that \( g \) is instantiated.

When the `instanceOf` edges of a graph \( g \) determine a node map which can be extended to a homomorphism into some interpretation \( f(g) \), we have a rather special kind of system state. Such a system state contains and instantiates its model, which in turn contains and instantiates its self-instantiating metamodel. Importantly, an object can navigate its `instanceOf` arrow(s) to find out what Class(es) it belongs to. It can then follow `classifierInv` edges to find out what Properties its Class has, and generally view the models description of itself and its world. This self-knowledge capability is sometimes called `reflection`.

The explicit instantiation edges also give us an opportunity to refine our interpretation function \( \gamma' \). When we defined \( \gamma' \), we derived the edges and nodes of the new graph from the sources and targets of the `classifier` and `type` edges of the old graph. A graph which contains the metamodel, and which has instantiation edges explicitly tells us what the nodes and edges of the dealgebraised graph should be. We define the edges of our dealgebraised graph to be the nodes with `instanceOf` edges into `Property`, and the nodes of this new graph to be nodes with `instanceOf` edges into `Classifier`.

For ill-formed models, it is possible that the `classifier` and `type` edges will not define total functions `source` and `target` on the edges of the new graph. Definition \[ \text{9} \] takes care to exclude these models.

Another modification is needed from our \( \gamma' \) interpretation function. The graphs our function will apply to will contain `instanceOf` edges, and these will need suitable images in the interpreted graph. It is Classifiers that are allowed to have instances. That is, \( x \xrightarrow{i} C \) is OK in a graph \( g \) iff \( C \xrightarrow{i} \text{Classifier} \) there. (We abbreviate `instanceOf` to just \( i \).) Also, note that Classifier is a Classifier, so we will want \( \text{Classifier} \xrightarrow{i} \text{Classifier} \) in the metamodel, and hence in every system state. Now, the nodes of the interpreted graph will be exactly the instances of Classifier from \( g \), so \( x \xrightarrow{i} C \) is OK in \( g \) iff \( C \) is a node in \( \gamma(g) \). Also, the `instanceOf` edges define a node map for which we need to find a compatible edge component. Thus the arrow \( x \xrightarrow{i} C \) must map to an arrow \( C \xrightarrow{i} \text{Classifier} \) in \( \gamma(g) \), and the arrow \( C \xrightarrow{i} \text{Classifier} \) in \( g \) must map to an arrow \( \text{Classifier} \xrightarrow{i} \text{Classifier} \) in \( \gamma(g) \). We see therefore that the desired behaviour is obtained by adding an `instanceOf` edge into `Classifier` from every node in \( \gamma(g) \). Consider a (bad) graph where something instantiates a non-Classifier. That is \( x \xrightarrow{i} y \) but there is no edge \( y \xrightarrow{i} \text{Classifier} \). In this case, there is nothing for the edge \( x \xrightarrow{i} y \) to map to. Indeed, there will be no node \( y \) in \( \gamma(g) \), and so the proposed node component is not into the right codomain, therefore it can not be extended to a homomorphism.²

²An alternative to arbitrarily adding all these `instanceOf` edges is to add three Properties to the
In the following definition, the edges which come from Property nodes are “marked” with a 0, and the added `instanceOf` are marked with a 1.

**Definition 9.** Let \( g = (\text{nodes}, \text{edges}, \text{labels}, \text{source}, \text{target}, \text{label}) \) be a labelled graph such that for each \( n \in \text{nodes} \) where \( n \xrightarrow{\text{classifier}} \text{s} \) and \( s \xrightarrow{i} \text{Classifier} \), a unique \( t \in \text{nodes} \) such that \( n \xrightarrow{\text{type}} t \) and \( t \xrightarrow{i} \text{Classifier} \), and a unique \( \ell \in \text{nodes} \) such that \( n \xrightarrow{\text{name}} \ell \) and \( \ell \xrightarrow{i} \text{String} \). Then we define

\[
\gamma(g) = (\text{nodes}', \text{edges}', \text{labels}', \text{source}', \text{target}', \text{label}')
\]

where

\[
\begin{align*}
\text{nodes}' &= \{ n \mid n \xrightarrow{i} \text{Classifier} \} \\
\text{edges}' &= \{ (n, 0) \mid n \xrightarrow{i} \text{Property} \} \cup \{ (n, 1) \mid n \in \text{nodes}' \} \\
\text{labels}' &= \{ \ell \mid (n, 0) \in \text{edges}' \land n \xrightarrow{\text{name}} \ell \} \cup \{\text{instanceOf}\} \\
\text{source}'(n, x) &= \begin{cases} s & \text{if } x=0 \\ n & \text{if } x=1 \end{cases} \\
\text{target}'(n, x) &= \begin{cases} t & \text{if } x=0 \\ \text{Classifier} & \text{if } x=1 \end{cases} \\
\text{label}'(n, x) &= \begin{cases} \ell & \text{if } x=0 \\ \text{instanceOf} & \text{if } x=1 \end{cases}
\end{align*}
\]

Let us now construct a reflective system state from the examples available to us. First, we make the self instantiation of the metamodel explicit, by adding instantiation edges to the CClite reading of its metamodel elements object diagram, shown in Figure 3.12 on Page 53. These new edges are almost given by the type assignments in the object icons of the diagram. The first exception is that we substitute Classifier for Association, Class and DataType, treating specialising classes as synonyms. (Chapter 4 will treat generalisation properly.) Secondly, we make each Property do double duty as a String.

\[
\begin{align*}
\text{Property} &\xrightarrow{i} \text{Classifier} \\
\text{Classifier} &\xrightarrow{i} \text{Classifier} \\
\text{String} &\xrightarrow{i} \text{Classifier}
\end{align*}
\]

Having distinct properties with the same name would also prevent us from using the Properties as Strings trick, and would therefore double the size of the metamodel. It would also deviate from the current official metamodel.
These edges are the node component of the label preserving homomorphism $mm\to\mathcal{P}(\gamma'(mm))$ we mentioned earlier, whose image is shown in Figure 3.14. Let us call the metamodel with the additional instantiation edges $rmm$ (reflective metamodel), and the node map given by these instantiation edges $n$. If we now check that each of the $instanceOf$ edges has an appropriate image, we see that we indeed have an edge component $e : edges(rmm)\to edges(\mathcal{P}(\gamma(rmm)))$ such that $(n, e, id_{labels(rmm)}) : rmm\to\mathcal{P}(\gamma(rmm))$ is a homomorphism. That is, $rmm$ is $(\mathcal{P} \circ \gamma)$-self-instantiating. We pause to define this idea a little more generally, for an interpretation function $f$ that takes each node to a set of nodes from the same graph.

**Definition 10.** Let $g$ be an instantiated graph, $f : Graphs\to Graphs$ a function where $nodes(f(g)) \subseteq \mathcal{P}(nodes(g))$, and let $n : nodes(g)\to\mathcal{P}(nodes(g))$ be given by $x \mapsto \{y \mid x \xrightarrow{i} y\}$. If there exists an $e : edges(g)\to edges(f(g))$ such that $(n, e, id_{labels(g)})$ is a label preserving homomorphism, then we say that $g$ is $f$-self-instantiating.

Now let us add the apple-in-a-basket model to this reflective metamodel. The CClite reading of its metamodel elements object diagram (Figure 3.9 on Page 43) is shown in Figure 3.15. The required instantiation edges are shown in Figure 3.16. Let $rm$ (reflective model) be the union of $rmm$ and these two graphs. Then $rm$ is also $(\mathcal{P} \circ \gamma)$-self-instantiating.

The “bare” system state will be the AA reading of the apple in a basket object diagram, which is the graph shown in Figure 3.4 on Page 40. We show this along with the additional instantiation edges (shown with label “i”) in Figure 3.17.
Figure 3.16: The instantiation edges of the apple-in-a-basket model

Figure 3.17: The apple-in-a-basket system-state, with its instantiation edges
The union of this graph and \( rm \) will be called \( rss \) (reflective system state), and is also \( (P \circ \gamma) \)-self-instantiating.

What we now have is a definition of a static modelling language, including its semantics. The framework assumes that both the models (syntax) and snapshots (semantics) of this language are graphs, and that the graphs which are part of this language are those that both contain the metamodel, and self-instantiate under some interpretation function. For the specific language we have developed, the metamodel is \( mm \) and the interpretation function is \( P \circ \gamma \).

This graph modelling language is both mathematically precise, and a near enough approximation of the MOF subset of UML to allow us to study UML’s syntax in Chapter 5. Before doing that though, it is necessary to revise our view of object diagrams to conform with the official UML definition [Obj07c].

3.6 Reconciliation of the Two Views

Seeing object diagrams as a variant of the traditional notation for graphs is a useful aid to understanding, and has helped us introduce systems and models. Now we must adjust our view to see them like class diagrams and other UML diagrams, as representing part of a model.

Each object icon in an object diagram represents an InstanceSpecification, which is “a model element that represents an instance” [Obj07c, §7.3.22]. The InstanceSpecification has a link classifier to the Classifier that it represents an instance of. A Slot [Obj07c, §7.3.48] represents a named value attached to an instance. It has links owningInstance to its InstanceSpecification, value to its ValueSpecification(s), and a link definingFeature to its Property. In an object diagram, the name = value entries in the object icons, and the link ends all represent Slots.

When a Slot represents a link end, the value is an InstanceValue [Obj07c, §7.3.23], and must be linked to an InstanceSpecification. We see no reason why the InstanceSpecification should need this ValueSpecifications intermediary. A minor simplification of the metamodel is therefore proposed. We will consider InstanceSpecifications and ValueSpecifications to be specialisations of Specification, a new metaclass of our own invention. We thus see another occurrence of the functor definition of a graph. Figure 3.18 shows our new metamodel fragment in graph form. (This is the decoded, post-dealgebraisation form, rather than the algebraic coded form that exists within reflective system states.) The original metamodel fragment can be seen on the bottom line, and the new copy of the graph defining graph is on the top line. We will see shortly that, for well defined models, the two vertical arrows form a graph homomorphism. Another way of saying this is that two squares within the diagram commute: one with owningInstance and (the horizontal) classifier, the other with value and type.

The metamodel elements represented by the object diagram of Figure 3.3 on Page 39 are shown in Figure 3.19. Applying a dealgebraisation analogous to the \( \gamma \) we employed on the class diagram metamodel elements, we obtain the graph
Figure 3.18: Augmented metamodel fragment as a graph of Figure 3.20. It is the same as the “bare” system state (Figure 3.17 Page 58), modulo the names of the instances and values.

The models of our slightly augmented modelling language are the graphs that can be mapped into that of Figure 3.18 by a label preserving homomorphism. A reflective model is the union of a model and the metamodel, with the node part of the homomorphism as instanceOf edges. The instances of Slot and Specification will make no difference to the \( \mathcal{P} \circ \gamma \) interpretation of such a reflective model. So, how are we to define the intended semantics of this object diagram part of the model?

In fact, it is an error to talk about the “intended semantics” here, because the Semantics heading for InstanceSpecification [Obj07c §7.3.22] describes several different possible kinds of statements which InstanceSpecifications can be used in. However, there is no indication of how these different kinds of statement are to be expressed in the abstract syntax. We are unable to see any way of doing so in the language as currently defined. Modal operators may be a solution, as we discuss in Section 5.4 in connection with Interactions [Obj07c §14.3.13]. For now, we will therefore focus on one reasonable interpretation of these elements and hence of object diagrams. We take an InstanceSpecification to be an assertion that an object satisfying its description exists in the system state. Thus an object diagram is taken to assert that a configuration of objects matching its depiction exists in every state of the system. Although this is stronger than most practical usages, we believe that those usages can only be properly formulated by adding modelling constructs for temporal operators, and with this addition, our interpretation would satisfy all practical requirements.

A class diagram describes what kinds of objects can exist in a system, and what kinds of links can exist between them. That is, it says that every object and every link in the system is of one of the types it declares. Object diagrams on the other hand assert that a certain configuration of objects and links exists. We treated the universal statement of the class diagram by asserting that its instances must map homomorphically into it. The existential statement of the object diagram can be treated dually, by insisting that its decoded form can be mapped homomorphically into the system state \( ss \).
Figure 3.19: Metamodel elements of the apple-in-a-basket object diagram (Figure 3.3 on Page 39).
Figure 3.20: Dealgebraisation of the object diagram metamodel elements

The node part of the homomorphism \( ss \rightarrow (P \circ \gamma)(ss) \) is determined by instantiation edges. There is a similar restriction on the homomorphisms \( f(ss) \rightarrow ss \) that will witness satisfaction of the object diagram: the Classifiers must map to themselves. That is, although the object names do not matter, if \( a : Apple \) occurs in an object diagram, the node that represents this in the model must map to an instance of Apple.

In the previous Section we identified a class of reflective system states, which contained the metamodel fragment and self-instantiated under our interpretation. Now, with an extended metamodel and a second interpretation function to express the existential semantics of object diagrams, these reflective system states can be characterised by the following formula.

\[
f(ss) \xrightarrow{id_{Classifiers}} ss \xrightarrow{i} (P \circ \gamma)(ss)
\]

Here \( f \) is the object diagram elements interpretation function, and the arrows indicate homomorphic extensions of their named (partial) functions. When these two homomorphisms exist, their composite is that given by the two vertical arrows of Figure 3.18. In other words, the valid system states are exactly the factorisations (under function composition) of that graph homomorphism.

Our aim was to reconcile the direct interpretation of object diagrams, with the view that they represent metamodel elements. But in fact we have now produced a weaker interpretation than the direct one. The model obtained from the object diagram can be satisfied by a huge range of system states, but the direct reading gives a system state that is unique, up to label preserving isomorphism. The homomorphism view allows several object icons to denote the same object, and
A major use of object diagrams is to describe metamodel elements. In this case, we want a specific model, not just a test for candidate models. The direct reading is the “largest” model without “extra” elements that satisfies the test.

We depict the situation in Figure 3.21, where our direct, naive view of object diagrams is shown as the dashed arrow. This is not a UML diagram, but merely our attempt to convey the relationships between the models and diagrams. In particular, the arrows are not links. We draw class and object icons on a UML note to refer to the icon itself (concrete syntax) rather than the model element it represents, as we did in Figure 2.1 (Page 23). That is, we use notes as quotation marks. This makes it clear how an object diagram and a class diagram can actually “mean” the same thing. The object diagram really represents model elements such as Instance-Specifications and Slots. This object model in turn, denotes a system state. Rather, we should say it is satisfied by a range of system states, but we have seen that there is a distinguished one among them. This system state is the model that the author of the object diagram intended to communicate. The same model is obtained by parsing the concrete syntax of some class diagram. So we see, the object and class diagrams do not “mean” the same thing at all. Rather the object diagram denotes what the class diagram represents.

Although this clarifies the role of the object diagrams which show metamodel

---

3This simple idea can be made complicated using category theory if desired. If we take the object model to be a single point “diagram”, then this largest image without “junk” it is the colimit, ie the image of the identity function. This is the same idea as “pushouts” discussed in Section 7.2 for gluing graphs together, except there, it is helpful.
elements, it is probably not the best way of thinking about them as they appear in Chapter 5. Rather, this analysis justifies simply taking the diagrams at face value.

3.7 Conclusion

We now have a semantics for very basic UML class and object diagrams sufficient for us to study the UML metamodel, and the syntax of particular models. Importantly, the semantics are derived from a careful study of the official definition, guided by the criteria we developed in Chapter 2. The semantic domain is labelled directed multi-graphs, and so is the abstract syntax. Each system-state contains its model and each model contains its metamodel. Instantiation is made explicit by specially named edges. The class model is interpreted by a “dealgebraisation” function and power graph construction. The system state satisfies it when the instantiation edges are the node component of a graph homomorphism into this interpretation. Dually, the object model is dealgebraised, and is satisfied when a class-preserving homomorphism exists into the system state.

We are not the first to propose that UML should be formally defined in terms of graphs. Our primary contribution is to argue for this based on a careful study of the official, informal definition. Our use of interpretation functions is original, so far as we know. Instantiation is often formalised as graph homomorphism \[ \text{[BH02, EHHS00, ZHG05]} \], but our introduction of interpretation functions has several advantages. It allows different parts of the model to be interpreted in different ways, reflecting the different diagrams types used to specify those parts. With an interpretation function, two or more levels of instantiation are possible. Without an interpretation function, every model is trivially self-instantiating, whereas we have shown that a fragment of the UML metamodel is self-instantiating under our definition, whilst models in general are not.

Another approach which can exhibit non-trivial self-instantiation is presented in \[ \text{[GFB05]} \]. There, Gogolla, Favre and Büttner “squeeze” the traditional four modelling levels “into a single object diagram”, in order to give precise definitions of terms such as “strictness” and “potency”, which are common in the metamodelling literature. In addition to explicit instantiation of nodes by nodes, second order edges connect edges to the edges they instantiate. Thus the structure they work with is not a traditional graph, but rather what they call a “layered graph”. After some discussion, they arrive at the notion of a “strongly well-typed” layered graph, which amounts to saying that the typing edges form graph homomorphisms. This proposal does not match UML’s notion of instantiation. With UML, edges at a lower level instantiate nodes at the next higher level. For example, links instantiate Associations, which are a kind of Classifier. The authors are not alone in accepting at face value the idea that object diagrams instantiate class diagrams, forgetting that in UML, the class diagram is an abstraction of the actual model, which is more accurately rendered as another object diagram. Our approach makes this abstraction explicit in the “dealgebraisation” interpretation function.
Graphs are a very expressive medium, but expressiveness can only be gained at the cost of analysability. The difficulty of analysing graphs is demonstrated by the problem of finding embeddings of one graph in another. This problem is known to be NP-complete, yet it will probably be involved in determining model instantiation and system state transition. It therefore seems likely that analysis of UML models as graphs will be computationally intractable. However, this is not a reason for abandoning graphs in favour of some other formalism. Anything else sufficiently expressive would face the same problem. It seems likely that UML itself is too expressive for automated analysis, and that any such analysis would require further abstraction. The formalised definition that we seek for UML would stand as a reference to better understand the results of analysing more abstract representations. In Section 7.3 we discuss to what extent the analysis of an abstraction can yield understanding of the original.

In Chapter 5 we will carefully study the syntax and semantics of a simple dynamic UML model. But first, in the next Chapter, we examine some of the more sophisticated and troublesome aspects of the class diagram notation used in the UML metamodel.
Chapter 4

Breaking the Symmetry of Association Ends

This chapter continues our study of UML’s “repository semantics” which we began in Chapter 3. The UML metamodel makes heavy use of the annotations \{union\} and \{subsets x\} in its association ends. There is no agreed precise semantics for associations, and many interpretations are possible. It has been noted [Obj, Issue #5977] that under some very plausible interpretations, the annotation \{unique\} is symmetrical. Symmetrical in the sense that if one association end is marked \{unique\} then the other end also has the property it signifies. It is undesirable for these annotations to be symmetrical. If they were intended to apply to all ends of an association, they would be attached to the association itself rather than to a particular end.

At the time of writing, discussion on the \{unique\} annotation in the OMG forum on UML just cited seems to have reached the conclusion that a proposed interpretation by Dragan Milicev [Mil06, Mil07] is a good solution to the problem. The naive view takes the annotation to be a constraint on what links in the system state can belong to the association. Milicev instead sees it as a parameter of the association navigation action ReadLinkAction [Obj07c §11.3.33].

Milicev does not address the \{union\} and \{subsets x\} annotations, which we will show also possess undesirable symmetry properties, even under reasonable interpretations compatible with his proposal. Adopting the principles at work in Milicev’s proposal, and applying them more uniformly, we achieve an interpretation which breaks the symmetry of these annotations as well.

Rather than focussing on ReadLinkAction and ReadStructuralFeatureAction, we frame the discussion in terms of the value of a models Properties, when they occur in a reflective system state. We find support in the official definition for viewing a Property as a function from the instances of its classifier to collections of values. Because of this, we must first develop a proper account of instantiation in the presence of generalisation, a task we deferred in the previous chapter. The actions are then defined in terms of the Property values, which can also be used
to define OCL navigation expressions, queries for model transformations and the like.

Our account of Property values is supported by a new kind of graph which we call a list-graph. This is like the edge-labelled directed multi-graphs we have already studied, except that instead of being a multi-set, the collection of edges with a given label leaving a given node, is a list.

The class diagram shown in Figure 4.1 will be used as an example throughout the chapter. We assume the Properties are owned by the Classes and not by the Associations. That is, we will use the model obtained by a CC reading (Chapter 3 Page 39). The other cases are similar. In the first Section, we exhibit the undesirable symmetry properties of the \{union\} and \{subsets x\} annotations under the obvious interpretation of associations and navigation. The second Section outlines Milicev’s solution and our generalisation of it, and shows that the undesirable symmetry of our examples disappears under this interpretation. We conclude with some reflections on the intuitiveness of this interpretation and some of its consequences.

4.1 Naive Symmetry

In Chapter 3 we suggested that the metamodel, models, system states and reflective system states are all edge-labelled directed multi-graphs. We did not consider object navigation expressions of the form \texttt{object.property}, for \texttt{object \in \texttt{nodes}} and \texttt{property \in \texttt{labels}}. These expressions occur in OCL, and textual concrete
syntax for the actions ReadLinkAction and ReadStructuralFeatureAction [Obj07c §11.3.33,11.3.37]. These are all best defined in terms of Properties.

[A] property represents a value or collection of values associated with an instance of one (or in the case of a ternary or higher-order association, more than one) type. This set of classifiers is called the context for the property; in the case of an attribute the context is the owning classifier, and in the case of an association end the context is the set of types at the other end or ends of the association. [Obj07c §7.3.44]

Thus, a Property is a function from its “context” to collections of values of its type. Unless the Property is owned by an association, its context is the instances of its classifier. When a Property is owned by an Association it would be sets of instances indexed by the labels of the other Properties of the Association. (The definition does not seem to recognise that we need to know which Property each object in the context collection belongs to.) This could be made precise, but for our work with the metamodel, we are only concerned with Properties owned by Classes.

When we write object.property then, we mean the value of the Property named property for the argument object. One might expect that these navigation expressions could be evaluated in our graph setting, by beginning at the node object and following the edges labelled property, and returning the nodes at the other ends of these edges.

\[ a.\text{bee} = \{ b \mid a \xrightarrow{\text{bee}} b \} \] (4.1)

This Section will show that such a straightforward definition of Property values is not a good idea, because it leads to unwanted symmetry. It also only caters for sets, whereas, UML has other kinds of collection as well. We defer discussion of these to the next section.

Figure 4.1 contains some new notation apart from the \{union\} and \{subsets \} annotations. The empty headed arrows indicate generalisation, or its inverse, specialisation. For example, \( A2 \) specialises \( A0 \) there. We encountered this in the previous Chapter, but avoided it by pretending that it merely defined Class synonyms. That was good enough for our purposes there, but now we must define it, because it affects what instances a Classifier has, and thus what context a Property has. Empty headed arrows as shown in Figure 4.1 represent Generalizations [Obj07c §7.3.20]. These model elements have a link to a general Class and a specific Class. Generalisation express UML’s version of the idea of inheritance found in most object oriented programming languages. “\( A1 \) specialises \( A0 \)” says roughly, that \( A1 \) is a kind of \( A0 \), like newspaper is a kind of publication. Publications have readers, therefore newspapers have readers. In UML, each Property defined for a Classifier is implicitly defined also for its specialisations.

We need to extend our metamodel fragment to handle this. Rather than work indirectly with the metamodel elements that must be interpreted by our “dealge-
We add a class Generalisation and a pair of Associations between it and Classifier with the following graph edges.

- **Generalisation** `instanceOf` **Classifier**
- **Generalisation** `special` **Classifier**
- **Generalisation** `general` **Classifier**

**Definition 11.** Let a model (graph) be given with nodes $C_0, \ldots, C_N$. If $g$ `instanceOf` $C_1$ and $g$ `special` $C_0$ we say that $C_1$ specialises $C_0$ and that $C_0$ generalises $C_1$. If $CN$ specialises $C(N-1) \ldots$ specialises $C_0$ for some $N \geq 0$ we say that $CN$ is a subclass (or subassociation or subclassifier) of $C_0$, and $C_0$ is a superclass (or superassociation or superclassifier) of $CN$.

In Chapter 3, we considered the `instanceOf` edges to completely determine the instances of a Classifier. This must now be reconsidered.

**Definition 12.** A node $a$ is an instance of a node $A$ if there is a subclassifier $A'$ of $A$ such that $a$ `instanceOf` $A'$. We sometimes write $a : A$ to indicate that $a$ is an instance of $A$.

Here, and throughout this Chapter, we assume a model which is like the one shown in Figure 4.1 except that there may be more than 2 pairs of classes like (A1,B1) and (A2,B2). We will use class names $A0, \ldots, AN, B0, \ldots, BN$, association names $R0, \ldots, RN$ and association end names $aye0, \ldots, ayeN, bee0, \ldots, beeN$. As in Figure 4.1 each $RN$ will be an association with ends $ayeN$ at $AN$ and $beeN$ at $BN$, and for each $N > 0$, we assume that $AN$ specialises $A0$ and $BN$ specialises $B0$. We sometimes write $A, B, R, aye, bee$ for $AM, BM, RM, ayeM, beeM$ for some arbitrary $M \leq N$. The Property which represents the association end $ayeN$ will be $P_{ayeN}$ and the model will have an edge $P_{ayeN}$ `name` $ayeN$.

The `{union}` and `{subsets x}` annotations are described in [Obj07c §7.3.44, Property associations subsettedProperty and isDerivedUnion]. In the abstract syntax, `{union}` is a boolean attribute `isDerivedUnion` of Property, and `{subsets x}`, is a reflexive Association `subsettedProperty` on Property. Thus we add two further edges and one node to our metamodel as follows.

- **Property** `isDerivedUnion` **Boolean**
- **Property** `subsettedProperty` **Property**

We could also extend the metamodel with nodes **Class** and **Association** and two Generalisations making these subclassifiers of Classifier, but these details are not
relevant for the property annotations we are considering. We will continue to see Class and Association as synonyms for Classifier.

The definition [Obj07c, §7.3.44, Property Semantics] says that a Property $y$ can be marked \{subsets $x$\} when “every element in the context of the subsetting property conforms to the corresponding element in the context of the subsetted property.” It is not clear what correspondence between elements this refers to. In every example we have seen in the metamodel, the subsetting Property is owned by a Classifier that specialises the owner of the subsetted Property. In these cases, the corresponding element could be the same element. There is some confusion in the UML definition about whether an object of a special class is an instance of its general classes [Obj07c, §7.3.20] or merely has a corresponding instance there, as the present citation suggests. Some readers consider that each Class has instances, and an object consists of one instance from each Class in a branch of the specialisation forest. We prefer to avoid needlessly multiplying our entities. Another part of the definition offers some clarification on what it means for one element to “conform to” another. “A Classifier conforms to itself and to all its ancestors in the generalisation hierarchy” [Obj07c, §7.3.8]. We assume it means the instances of the Classifiers rather than the Classifiers themselves. Thus one case where subsetting is fairly clearly permitted is in CC interpretations of “squares” like our $A_0, B_0, AN, BN$ where $ayeN$ can subset $aye0$ and $beeN$ can subset $bee0$. Fortunately, all the occurrences we have seen in the metamodel are of this form.

When a Property $y$ is marked \{subsets $x$\}, that is $y \xrightarrow{\text{subsettingProperty}} x$ in the model, then “the collection associated with an instance of the subsetting property must be included in (or the same as) the collection associated with the corresponding instance of the subsetted property” [Obj07c, §7.3.44, Semantics]. Here we must assume that the authors mean the instances of the owner of the Property rather than instances of the Property. Definition 13 makes explicit what we take this passage to mean, where we temporarily take Equation 4.1 (on Page 68) to define the dot notation. The \textit{subsettingProperty} link between the Properties is taken to be a constraint on the valid system states, so the definition says which system states “satisfy” the link.

\textbf{Definition 13.} Let a graph be given where

\[
\begin{align*}
e &: P_{bee} \xrightarrow{\text{subsettingProperty}} P_{bee0} \\
P_{bee} &\xrightarrow{\text{classifier}} A \\
P_{bee} &\xrightarrow{\text{name}} bee \\
P_{bee0} &\xrightarrow{\text{name}} bee0.
\end{align*}
\]

We say that $e$ is satisfied in that graph iff $(\forall a : A) a.bee \subseteq a.bee0$.

Now let us see an example of the unwanted symmetry that arises under this plausible view of Associations.
Example 1. Suppose we have a reflective instance of the model shown in Figure 4.7 satisfying its subsettedProperty edges. Suppose also that \( b \in a.\text{bee}1 \). Then we have

\[
\begin{align*}
\text{Equation 4.1} & : & b \in a.\text{bee}1 & \Rightarrow & a \xrightarrow{\text{bee}1} b \\
\Rightarrow & & b \xrightarrow{\text{aye}1} a & \text{partner edge} \\
\Rightarrow & & a \in b.\text{aye}1 \\
\Rightarrow & & a \in b.\text{aye}0 & \text{aye}1 \{\text{subsets ay}e0\} \\
\Rightarrow & & b \xrightarrow{\text{aye}0} a & \text{Equation 4.1} \\
\Rightarrow & & a \xrightarrow{\text{bee}0} b & \text{partner edge} \\
\Rightarrow & & b \in a.\text{bee}0 & \text{Equation 4.1}
\end{align*}
\]

and we thus have that \( a.\text{bee}1 \subseteq a.\text{bee}0 \), even though this is not explicitly declared on the class diagram nor in the model.

This is undesirable, but it is not quite the same as having \( \text{bee}1 \) marked \{\text{subsets bee}0\}, because of the interaction of this annotation with \{\text{union}\}. When an end is marked \{\text{union}\}, this “means that the collection of values denoted by the property in some context is derived by being the strict union of all the values denoted, in the same context, by properties defined to subset it” [Obj07c, §7.3.44, Semantics]. That is, if an end \( \text{aye}0 \) is marked \{\text{union}\}, that is \( P_{\text{aye}0} \xrightarrow{\text{isDerivedUnion}} \text{true} \), then the usual definition of the value of that Property is overridden by something like the following.

\[
b.\text{aye}0 = \bigcup \left\{ P_{\text{aye0}} \mid P_{\text{aye}} \xrightarrow{\text{subsetsProperty}} P_{\text{aye0}} \right\} \tag{4.2}
\]

Equation 4.1 fails with this equation in place, because now \( a \in b.\text{aye}0 \) does not entail \( b \xrightarrow{\text{aye}0} a \). However, in the absence the \{\text{union}\} annotation on \( \text{aye}0 \), the unwanted symmetry of \{\text{subsets x}\} remains.

Ends marked \{\text{union}\} are read-only [Obj07c, §7.3.44, Constraint 8], presumably because anything added there would also have to be added to at least one of the subsetting ends, but there is no systematic way of saying which ends the value should be added to. Such unpredictable effects are clearly undesirable. Also, removing a value here would require the value be removed from all subsetting ends, which seems somewhat heavy-handed.

Example 2. Our current naive interpretation of associations also gives \{\text{union}\}
an unwanted kind of symmetry. Assume that $b \in a.\text{bee}0$, then

$$b \in a.\text{bee}0 \Rightarrow a \xrightarrow{\text{bee}0} b$$
$$\Rightarrow b \xrightarrow{\text{aye}0} a$$
$$\Rightarrow a \in b.\text{aye}0$$
$$\Rightarrow a \in b.\text{aye}N \text{ for some } N \text{ that } \{\text{subsets } \text{aye}0\}$$
$$\Rightarrow b \xrightarrow{\text{aye}N} a$$
$$\Rightarrow a \xrightarrow{\text{bee}N} b$$
$$\Rightarrow b \in a.\text{bee}N$$

from which we may conclude that $\text{bee}0$ is the union of ends whose opposites subset $\text{aye}0$. Again, the annotation is working on the whole association and not just the end.

Even worse, if $b \in a.\text{bee}2$ for some $a$ and $b$, then $b \in a.\text{bee}0$. If our model has only the 6 classes shown in Figure 4.1 then $\text{aye}1$ is the only end that $\{\text{subsets } \text{aye}0\}$, so as we have just argued, $b \in \text{bee}1$. Thus for every $b \in B2$ which participates in the association $R2$, we have $b \in B1$. We pointed out on Page 49 of Chapter 3 that UML allows an object to belong to several distinct classes, even when these classes are not related by generalisation. Here we have a situation where only these “hybrid” objects can participate in an association.

**Example 3.** Assuming that Figure 4.1 shows all of the classes in the model, the only objects that can be linked across $R2$ are “hybrids” $b$ such that $b : B1$ and $b : B2$. Non-hybrids are unlinkable there.

**Example 4.** Suppose we find a linkable pair $(a, b)$ that are not linked across $R2$ or $R0$. The pair are not linked across $R1$ either, because $\text{aye}1 \{\text{subsets } \text{aye}0\}$. If we now link them across $R2$, a corresponding link must appear in $R0$, and hence in $R1$ as well. If on the other hand, the model contained classes $\text{AN}$ and $\text{BN}$ associated by $\text{RN}$, with $\text{aye}N \{\text{subsets } \text{aye}0\}$, the new link could appear in $R1$ or $\text{RN}$. Either way would satisfy the constraints, but there is no principled way of determining where the link should show up.

This extra level of non-determinism is what we believe the UML’s designers were attempting to rule out with their prohibition on updating associations with an end marked $\{\text{union}\}$. Indeed, since our reasoning above shows that updating $R2$ entails updating $R0$, perhaps we should conclude that $R2$ is also read-only.

The common assumption that has lead to all these bad consequences is that the value at an association end is determined by the links belonging to the association. Equation 4.1 is a concise statement of this assumption. We should not be surprised by these consequences though. Links have two ends, so we should expect their presence in an association to have symmetrical effects. If we want association end annotations to properly belong to the ends, we require a more sophisticated account of Property valuation.
4.2 Derived Property Values

We have seen that an obvious way of defining Property values leads to unwanted symmetry of association end annotations. Our solution to this problem extends the work of Dragan Milicev [Mil06, Mil07]. Milicev tackles a related problem with the association end annotations \{unique\} and \{ordered\}, which govern what kind of collection one obtains when reading values from that end. We begin by outlining the theory of collections then explain Milicev’s solution. We generalise his approach and apply it to our graph theoretic setting, introducing a new kind of graph which captures the required collection structure. The value of Properties is then defined, and the misbehaviour seen in the examples of the previous section are shown to disappear.

The 4 kinds of collection in UML are lists, bags, ordered sets and sets. These are type constructors rather than types, since for example a list is always a list of integers or Apples or some other type. Lists and sets are obvious enough ideas. A bag (also known as a multiset) has no order, but can contain the same element several times. An ordered set (totally ordered) only contains each element once, but the elements are arranged as a first element, a second and so on. A multiset can be obtained from a list by “forgetting” about its order, and an ordered set can be obtained from a list by forgetting duplicate elements. If you forget the order and remove duplicates, the list becomes a set. In fact it becomes the same set regardless of what you choose to forget first. Figure 4.2 shows these relationships between collections. We now set out these ideas more carefully.

Definition 14. We write $\text{ISN}$ for the set of initial segments of the natural numbers $\{\{\}, \{0\}, \{0, 1\}, \{0, 1, 2\}, \ldots\}$. A list over a set $U$ is a function $N \rightarrow U$ for some initial segment $N \in \text{ISN}$.

Duplicates in a list $\ell$ can be forgotten by taking the first occurrence of each element. The following equation defines a function which takes a list, and yields a function that tells what position $n$ the first occurrence of each element $x$ is in that list.

$$\text{posfirst}(\ell) = \{(x, n) \mid x \in \text{range}(\ell) \& n = \min \{m \mid (m, x) \in \ell\}\}$$
From this we obtain an ordered set by letting \( x \leq y \iff \text{posfirst}(\ell)(x) \leq \text{posfirst}(\ell)(y) \).

The order of \( \ell \) can also be forgotten to obtain a bag or multiset. A multiset is a function \( c : U \rightarrow \mathbb{N} \), where the number \( c(x) \) indicates how many occurrences of \( x \) it contains.

\[
c = \{(x, n) \mid n = \text{size}\{m \mid (m, x) \in \ell\}\}
\]

Forgetting the duplicates of this multiset yields \( \{x \mid c(x) \geq 1\} = \text{range}(\ell) \).

We call a bag which actually contains duplicate elements a proper bag. Under the naive interpretation of associations discussed in the previous section, \( a\text{.bee} \) is a proper bag if the association \( \text{bee} \) is a proper bag iff \( b\text{.aye} \) is a proper bag. That is, for some \( a \) there is some \( b \) for which we have more than one edge \( a \xrightarrow{\text{bee}} b \). Since these edges always come in opposing pairs, there is also more than one edge \( b \xrightarrow{\text{aye}} a \). If one end is marked \( \{\text{unique}\} \), values at both ends are always sets.

The essence of Milicev’s solution to this unwanted symmetry problem is to decouple the contents of the association from the values obtained by reading its ends. An association for Milicev, is always a bag of links. If Property attribute \( \text{isUnique} \) is true, then duplicate values in the collection are forgotten before the collection is returned. He also solves a related problem with ends marked \( \{\text{ordered}\} \) by attaching ordering annotations to the object references in the links. The specified capabilities of the link actions CreateLinkAction and DestroyLinkAction \([\text{Obj07c}, \S 11.3.14,11.3.17]\) for associations with an ordered end can not be achieved when the association is a list of ordinary links.

We would like a graph-like structure that can naturally handle UML’s ordered Properties. It seems unfortunate to have an ad-hoc semantics just for those associations with the \( \{\text{ordered}\} \) annotation at one of their ends. We would like to see uniform semantics, with the end annotations merely indicating which forgetful functor should be applied. Recall that uniformity is an aid to understanding, and understandability is lacking in the current definition of UML. Ordering the \( \text{links} \) is not a solution, because that also entails symmetry of the ordering at the association ends. However, we can achieve the desired uniformity by ordering the outgoing \( \text{edges} \) in our graphs.

Definition 1 (on Page 39) defines a directed multi-graph as a set of edges, a set of nodes and a pair of parallel functions, \( \text{source} \) and \( \text{target} : \text{edges} \rightarrow \text{nodes} \). Let us consider alternative graph structures and ways of defining them. Ordinary (non-multi) directed graphs can be defined as a relation over a set of nodes, \( G \subseteq \text{nodes} \times \text{nodes} \), or equivalently as a function which takes each node to the set of nodes at the other ends of its outgoing edges, \( g : \text{nodes} \rightarrow 2^{\text{nodes}} \). Labels can be introduced by making \( g \) a label-indexed family of functions \( (g_\ell : \text{nodes} \rightarrow 2^{\text{nodes}})_{\ell \in \text{labels}} \). Directed multi-graphs can be similarly defined by replacing the node sets with multisets. A pattern is emerging, suggesting that the nodes can be mapped to any collection of nodes. In the following definition, we take this pattern to its conclusion, where each label maps each node to a list of nodes.

74
Definition 15. A labelled directed list-graph \( g \) is a label-indexed family of functions from the nodes to lists of nodes:

\[
\left( g_{\ell} : \text{nodes} \rightarrow \bigcup_{N \in \text{IS}} \text{nodes}^N \right)_{\ell \in \text{labels}}
\]

It is desirable to define these kinds of graphs as functors, as we saw in Section 3.1, so that the general theory of graph transformation systems \([\text{EEPT06}]\) and other category theoretic machinery can be applied. We have not yet found a way of doing this. Also our power-graph construction of Definition 7 (on Page 49) does not apply directly to list-graphs. The order forgetting functor must be applied to each node value first to obtain a multi-graph.

We let reflective system states be list graphs, and are now ready to define the value of a Property \( P \) for a member \( x \) of its context, in a system state \( s \). For a system state \( s \) and edge label \( \text{name} \), the function \( s_{\text{name}} \) takes each node to a list of nodes. We will use this notation in what follows. The Property value function and an auxiliary function for the subsetted values are defined by mutual recursion. This replaces Equation 4.2 of the previous section, which only handled sets. The recursive definition depends on there being no cycles or infinite chains of edges.

**Definition 16.** Given a reflective system state \( s \), a node \( P \) where \( P \xrightarrow{i} \text{Property} \) and \( P \xrightarrow{\text{classifier}} X \) for some \( X \xrightarrow{i} \text{Classifier} \), and a node \( x : X \), we define

\[
\text{subsettedVals}(P, x) = \bigoplus_{P' \in \text{subsettedPropertyInv}(P)} F_{P'}(\text{value}(P', x))
\]

where, \( F_{P'} \) is the forgetful functor from the collection type of \( P' \) to the collection type of \( P \), and \( \bigoplus \) is either list concatenation, multi-set union, concatenation followed by removal of duplicates, or set union, depending on the collection type of \( P \), and

\[
\text{value}(P, x) = \begin{cases} 
\text{subsettedVals}(P, x) & \text{if } P \text{ is Union} \\
\text{subsettedVals}(P, x) \oplus F_P(s_{\text{name}}(P)(x)) & \text{otherwise}
\end{cases}
\]

where \( F_P \) is the forgetful functor from lists to the collection type of \( P \).

This only works if each subsetting property is at least as structured as the subsetted property. Otherwise the required structure must be “invented” before the subsetted property value can be given. For example, the subsetting property might be set-valued, but the subsetted property list valued. So, in what order do the elements of the set appear? Constraint [5] in the definition of Property concerns the strength of types in subsetting relationships, but this does not account for collections, only the type of values in the collections.
Notice also that we rely on the subsettedProperties of a given Property to have a specific order, which we use to correctly assemble the resulting list. There is actually no indication in the definition of Property [Obj07c §7.3.44] or the metamodel diagram [Obj07c Figure 7.12] that either end of the association subsettedProperty is ordered. The opposite end should be, because without this, it is undetermined which order the lists should be concatenated to yield the value of the subsetted Property.

The actions ReadStructuralFeatureAction and ReadLinkAction can now be defined in terms of this property valuation function. Action execution will be considered carefully in Section 5.6 here we just outline how it will work. Consider a system state in which a ReadStructuralFeatureAction [Obj07c §11.3.37] is enabled (ready to execute). A model fragment corresponding to the expression a. bee is shown in Figure 4.3. The ReadStructuralFeatureAction has a link object to a Pin ([Obj07c §11.3.28], again see Section 5.6) containing the value a and a link structuralFeature to the Property bee. Execution of the action will result in value(bee, a) being placed on the OutputPin.

Now let us reconsider our examples from the previous section in the light of our new definition of Property values. Example 1 fails because $a \in b.aye0$ no longer entails that $b \xrightarrow{aye0} a$. The value is in $b.aye0$ because of a link in $R1$, not one in $R0$. Equation 4.2 already made this so in the presence of the \{union\} annotation, but Definition 16 does the job even without it.

The very first step fails in Example 2 because bee2 \{subsets bee0\}, and could thus be the source of the $b$ in $a.bee0$. This possibility is enough to justify the \{union\} belonging to an end rather than the association, but the situation here warrants further investigation.

Let us consider a model represented by only the top square in Figure 4.1 Then $a \xrightarrow{bee0} b$ does follow from $b \in a.bee0$. We noted before that CreateLinkActions and DestroyLinkActions are not allowed for associations with a \{union\} end, like our $R0$. We should not conclude from this that $R0$ must be empty however. What-
ever \( R_0 \) contained in the initial system state, it will contain forever. Computer programmers concerned with the initialisation of their systems are sometimes lead to the fallacy that the initial state of a system is always empty. In fact, since all action in UML systems is carried out by objects \[\text{Obj} \{ \text{§6.3.1,The Basic Premises}\}, \]

an empty system state can never evolve into a non-empty one. We should also remember that UML is moving beyond its software development origins, and becoming a general purpose language for describing systems of distributed objects which evolve in discrete steps.

We therefore allow the possibility that \( a \xrightarrow{\text{bee}} b \). Because evaluation of \( b.\text{aye}() \) is now defined by Definition \[16\] and not Equation \[4.1\] it does not follow that \( a \in b.\text{aye}() \). That is, the \{union\} annotation actually hides the \( R_0 \) links from the annotated end \( \text{aye}() \). Thus “preexisting” links in \( R_0 \) are a second source of asymmetry for this \{union\} annotated association.

Examples \[3\] and \[4\] depend on reasoning in Example \[2\] which we have shown no longer works under our new definition of association end navigation. Any pair \((a, b)\) with \( a \in A_2 \) and \( b \in B_2 \) can be linked across \( R_2 \) without any impact on the contents of \( R_0 \) or \( R_1 \). Similarly, changes to the contents of \( R_1 \) need not have unpredictable consequences elsewhere.

### 4.3 Conclusion

Some UML users might be surprised to find that we can have \( a \in b.\text{aye} \) yet \( b \notin a.\text{bee} \). Without this possibility though, \{union\} and \{subsets \( x \)\} annotations can not properly belong to association ends, but instead affect all ends of the association.

We suspect that many UML features have consequences which clash with the intuitions of some UML users. The current definition of UML makes it difficult or impossible to determine what these consequences are. Our attempts to clarify the consequences of a couple of minor language features has cost us a great deal of effort, mostly trying to find a precise formulation which agrees with as much as possible of the official definition. A precise official definition of the language semantics would enable theoretical workers to expose these consequences much more easily.

We believe that using UML to model a preexisting or required system can improve stakeholder understanding and agreement, and thus improve project outcomes. The quality of the understanding and agreement achieved is limited by the understanding of the language itself. Because of its visual notation and familiar concepts, UML is often seen as “easy to understand” compared with typical formal languages. However, the extent to which it can be understood is limited by the precision of its definition.

In any case, we now have a precise enough definition of the semantics of the subset of UML used to define UML’s abstract syntax. List graphs are the fundamental structure used as system states, models and metamodels, and the values represented by Properties in reflective system states is now defined in a way that
makes good sense of the association end annotations. In the next chapter we study a small model which uses some of UML’s language of system dynamics. The semantics we have developed here will serve to make the syntax precise, and also as a foundation for the dynamic semantics of that model.
Chapter 5

Testing the Definition

The purpose of this chapter is to test the adequacy of the official definition of UML [Obj07c]. The test we perform is to take a small example model, and see whether we can infer its meaning by referring to the definition. In particular, the example model appears to be inconsistent, so we wish to determine whether this inconsistency is realised in the meaning that the definition gives us. The example model is presented in Section 5.1. It is inspired by an example of Stephen Mellor [HS05]. This test leads us to a thorough investigation of key parts of the UML definition. As a result, we develop an understanding of the dynamic aspects of UML, and identify several conceptual flaws.

From Section 5.2 we begin parsing the model’s concrete diagrams into UML’s abstract syntax. This gives us the model as metamodel elements whose meaning can be looked up in the definition. As a result of our work in Chapter 3 we can show this abstract syntax using UML object diagrams without loss of precision. Section 5.2 discusses the UML definition document [Obj07c] and outlines the task of parsing the diagrams and finding the meaning of the model elements. The class diagram from the example model is parsed, finding many details of the abstract syntax omitted in our preliminary work in Chapter 3. Section 5.3 begins our study of UML’s dynamics, introducing the notions of behaviour and event. This first approach to UML’s dynamics confronts many conceptual difficulties and distinctions. In Section 5.4 we parse and interpret the example sequence diagram, whose meaning is defined in terms of event occurrences. The section uncovers an apparent UML bug. Object behaviour and how it is caused is the subject of Section 5.5 and the state machine diagram serves as our example of this. The penultimate Section 5.6 looks at UML activities and actions, and in particular the entry action of one of the state machine states. This Section also introduces graph transformation and outlines how the prescriptive parts of the dynamic semantics could be precisely defined using this technique. The concluding Section 5.7 draws together the findings of the Chapter to see whether the consistency of the model can be determined from the UML definition.
5.1 An Apparently Inconsistent Model

This section presents our formulation of an example model due to Stephen Mellor, which we shall use throughout this Chapter, and also in Chapter 6. The point of this example is that it appears inconsistent, yet, Mellor claims, the definition of UML does not detect this. The quote below is from an interview, and Mellor allows himself a little humour and informality in making his point.

Consider this. We have two state chart diagrams, one of which sends a single signal $X$ to the other. In the same system, we have a sequence diagram that shows lifelines for two objects whose behaviour is captured by the state chart diagrams, one of which sends a single signal $Y$ to the other. Both diagrams are intended to describe the same behaviour; that is, a single message being sent between them. Which of these two - contradictory - models is correct? Astonishingly, UML’s answer is Yes. So long as the syntax of each of the two diagrams is correct, UML is cool.

-Stephen Mellor in [HS05]

There certainly should be some definite meaning attached to UML diagrams, so that tools can detect that no system can satisfy all the diagrams in Mellor’s example. Is he correct in claiming that the definition gives us no more than the languages syntax? He is surely aware that a large part of the document is written under subheading “semantics”, but the implicit claim is that this text fails to clearly define what happens when a given model is executed.

Part of Mellor’s question can be answered immediately. He asks “Which of these two - contradictory - models is correct?” That is, if there really is a contradiction, is it the state machine or the sequence diagram that determines what will happen. We have distinguished between description and prescription. A description can be true or false, depending on whether things are as described. Prescriptions such as mathematical definitions and computer programs, can not be false, they actually make the world the way they describe. UML contains both kinds of statement. The state machines are prescriptive, because object behaviour is directly defined by them. Sequence diagrams on the other hand describe something that may or may not happen, indeed, may or may not be possible. From the system’s point of view then, the state machine is “correct”. However, sequence diagrams are often used to describe user requirements, in which case it is the state machine that must be changed.

We choose to formulate the problem in the style of Executable UML [MB02], including a class diagram, and omitting the arbitrary second state machine. A class diagram (Fig. 5.1 Page 81) declares the two classes. The association between them will be used to target the signal.

We only give one state machine diagram (Fig. 5.2 Page 81), which describes the behaviour of class $A$, including the send action.

80
Finally, the sequence diagram (Fig. 5.3, Page 81) shows an object of class \( A \) accepting a \( W \) signal from an Actor (or “external entity”), entering state \( s' \) and (erroneously?) sending a \( Y \) signal to a \( B \) object, presumably its \( \text{ex} \).

5.2 The Definition of UML and our Class Model

This section introduces the official UML definition document \cite{Obj07c} which we will refer to even more frequently in this chapter than we have in previous ones.
As a simple introductory example of the kind of task undertaken in subsequent sections, we parse the class diagram of our example model, Figure 5.1 (on Page 81). The difference between this task and the class diagram parse we undertook in Chapter 3 is that here, we use the full UML metamodel rather than a tiny fragment of it.

UML is a huge language. Most applications only require a subset of it, and most tools only support a subset of it. This tends to lead to a situation where only the intersection of all these subsets is portable. In order to maximise the portability whilst providing useful modest subsets of the language, UML has been organised into 4 increasingly inclusive compliance levels, 0 to 3 [Obj07c §2.2]. The language is also organised into language units which are approximately disjoint sub-languages such as the state-machine and activity sub-languages. These language units are themselves broken into increasingly inclusive language increments.

To achieve all of this, the definition consists of many packages which are merged [Obj07c PackageMerge, §7.3.40] in a variety of configurations to produce these various sub-languages and compliance levels. In what follows, we will assume the full language: UML compliance level 3.

Chapters 3 and 4 were intended to prepare us for the present Chapter by making it clear what are the instances of the UML metamodel, but we did not consider the meaning of PackageMerge. Taking account of PackageMerge does not add any complexity to our semantic domain and abstract syntax however, because it is defined as a syntactic abbreviation. “A package merge between two packages implies a set of transformations, whereby the contents of the package to be merged are combined with the contents of the receiving package.” [Obj07c PackageMerge §7.3.40, Semantics]. The definition is at pains to point out that the transformation need not actually occur in any metamodel implementation (thus the emphasised “implies”), but makes it clear that the meaning of a model in which PackageMerges occur is equivalent to a model without PackageMerges, obtained from the original by transformation.

The “technical content” of the definition begins with Chapter 7 [Obj07c §6.5]. From that point till the Annexes, each chapter covers a subset of the UML metamodel elements, often coinciding with a diagram type. These chapters consist of a brief introduction, some class diagrams showing the part of the metamodel covered, a subsection on each of the metaclasses in those diagrams and finally a diagrams subsection. Each of the metaclass subsections includes generalisations, a description, attributes and associations, semantics and notation for that metaclass. In this Chapter, we will be referring to the notation parts of the definition in order to determine what metamodel elements our model diagrams represent. We also collect the information from the generalisations section of each metaclass to show how these model elements fit into the hierarchy of UML ideas, described in the definitions semantics entries. The diagrams subsections of the definition include tables summarising the graphical elements described in the notation passages of the metaclass descriptions. It is these tables which will guide the parsing of the diagrams of our example model.
We parsed an extremely simple class diagram, like that of our current example model, back in Chapter 3. We obtained the configuration of objects shown in Figure 3.9 (on Page 43) which included Classes, Associations and Properties. By consulting the table of graphic nodes in structure diagrams [Obj07c, Table 7.2, §7.4] we see that the class icons represent Classes. No surprises there. Similarly the graphic paths table [Obj07c, Table 7.3, §7.4] tells us that the line between the classes represents an Association. Although we have found no explicit statement that attributes and association ends in the diagram represent Properties, this is clear from the relevant definition sections. So, the model elements represented by our class diagram will belong to the same metaclasses as we saw in Chapter 3.

At that stage, we were working with a tiny fragment of the UML metamodel, so the resulting model was simple, with very few attributes and links. Now we are faced with the full metamodel, so what attributes and links will the Class, Association and Property objects have? To determine this for the Classes, we must locate all the different versions of Class in different packages in the definition. Each section defining one of these versions lists the attributes and associations and describes the semantics. But each of these versions of Class are also specialisations of other metaclasses, and so they incorporate the attributes associations and semantics of these more general metaclasses. The generalisation hierarchies of two of the four versions of Class are shown in Figures 5.4 (on Page 84) and 5.5 (on Page 84). These four different versions of Class are “merged” [Obj07c, PackageMerge, §7.3.40] to form the metaclass Class which our classes will instantiate.

Syntactically, the generalisations and the package merges just accumulate attributes and associations. It is complicated, but clear enough. It is not so clear how a coherent meaning is to be extracted from the generalisation and merge structure of the dozen or so short texts describing the semantics of these elements.

Association is defined only once in the definition, and most of its generalisation hierarchy is above Classifier, which is already shown in Figure 5.4. The remainder of the generalisation hierarchy is shown in §5.6 (on Page 85). Each Association is linked to at least two Properties called its memberEnds. The types of these association ends are the Classes that the Association associates.

The metaclass Property is defined four times in the definition. The generalisation hierarchy for the main one of these is shown in §5.7 (on Page 85).

Gathering associations and attributes from these hierarchies, we find that the class diagram of our example model represents the model elements shown in Figure 5.8 (on Page 86). Not all attributes are shown, because there are too many. For example, Property has over 20 attributes. Many of the association ends in the metamodel are marked \{subsets\...\}, where \...\ is some other association end we call the subsetting property. We discussed this annotation at length in Chapter 4 where we decided that it is an indication to the navigation expression evaluator to include the values of this property when evaluating the subsetting property. Definition 16 (on Page 75) states this formally. We indicate subsetting by placing several end names at the ends of the link paths in the diagram. Although our purpose in Chapter 4 was to make it possible for these \{subsets\...\} annotations to properly
Figure 5.4: Generalisation hierarchy of Class from Kernel package showing Figure references in [Obj07c]

Figure 5.5: Generalisation hierarchy of Class from Communications package [Obj07c §13.3.8]
Figure 5.6: Generalisation hierarchy of Association, showing Figure references in [Obj07c]

Figure 5.7: Generalisation hierarchy of Property, showing Figure references in [Obj07c]
Figure 5.8: Metamodel elements of the class diagram

apply to one end rather than being symmetric, they are often used in the metamodel in pairs, one on either end of an association. We have indicated these pairings by putting each name in the same position as its partner at the other end. Where no partner name exists, we indicate this by putting a `-' in its place. We have shown the values of some derived attributes, using the usual `/`-prefix notation. The association end Properties are owned by the end classes rather than the Association. As we discussed in Chapter 3, when no ownership dots [Obj07c, §7.3.3, notation] are shown in a diagram, any uniform reading is permissible.

The complex packaging and generalisation structure of the definition makes it a very time-consuming and error-prone task to determine exactly what any given model element consists of, and what the resulting element means. Our parse of the example model was executed initially without any tool support, just by reading the definition. We later checked the model by entering it in the open source UML 2.1 implementation of the Eclipse Modelling Framework [Ecl07]. None of this was easy, and we can not be completely confident that this parse is “correct”. This task
has been referred to as “navigating the metamuddle” \cite{FGDTS06}. That paper and our discussions with other researchers confirm that our experience is a common one.

5.3 UML Dynamics

In Chapter 3, we developed a tentative semantics for the static fragment of UML used in the UML metamodel. This has enabled an unambiguous account of the abstract syntax. The example model we are studying in this chapter includes static elements such as Classes and Associations, but also employs dynamic parts of the language, such as Events, Behaviors and States. This section develops an overview of the dynamic aspects of UML in preparation for the detailed work of the next few sections.

The main ideas of UML’s dynamics are best introduced by some key quotes from the definition. “A model contains three major categories of elements: Classifiers, events, and behaviors.” \cite[§6.4.1]{Obj07c}. A classifier “describes a set of instances that have features in common” \cite[§7.3.8]{Obj07c}. “An event is the specification of some occurrence that may potentially trigger effects by an object.” \cite[§13.3.13]{Obj07c}. “Behavior is a specification of how [the instances of] its context classifier change[s] state over time.” \cite[§13.3.2]{Obj07c}. Classifiers are the major element for describing the statics of the system, whilst events and behaviors describe the dynamics.

The words “behavior” and “event”, and much of the description in the definition leads us to think that behavior is about prescriptive dynamics, whereas events are descriptive. This distinction between description and prescription was introduced in Section 1.3. StateMachines and Activities are kinds of Behavior, and these are a kind of programming, instructing the objects how to behave. Perhaps surprisingly, Behaviors are also a kind of Classifier. Whenever a behavior is executing, it actually exists in the system state as an instance of the Behavior, often called a behavior execution. We will examine behavior executions more carefully in Section 5.4.

In our example model, Events are used in the sequence, and also as the transition trigger in the state machine. Figure 5.9 (on Page 88) shows the generalisation hierarchies of the kinds of Event represented by the example sequence diagram. In both kinds of usage, Events play the role of a condition of the form “such-and-such happens”. In the state machine, it is “when such-and-such happens, make this state transition”. In the sequence, the events are combined in a temporal order to describe a more complex happening. Reggio and co-authors suggest that temporal logic is needed for a formal treatment of sequence diagrams \cite{RCA01}. We pursue this suggestion in Chapter 6 (published as \cite{O’K06a}). However this view of Events

\footnote{Even the authors of the definition occasionally fail to distinguish between a classifier and its instances. The square brackets [] show what is clearly intended.}
and sequence diagrams as temporal conditions is not supported by a more careful reading of the definition.

Sequence diagrams represent Interactions [Obj07c §14.3.13]. These are not as we might expect, a kind of compound Event, but a Behavior. The generalisation hierarchy of Interaction can be seen in Figure 5.10 (on Page 89), along with those of the required OccurrenceSpecifications, which we will discuss shortly. An Interaction is not a behavior of any particular kind of object, but an “emergent” behavior in which several objects may participate. Behavior is a Classifier, so like the behavior of a particular object, the instances of an Interaction exist at run-time. For the prescriptive dynamics of an object, this is fine. The Behavior instance is like an executing process in an operating system, it has a clear identity that can be traced through time. It exists while the process is executing, and ceases to exist when the process terminates. An Interaction can not be like this. The execution of a UML system is non-deterministic. The beginning of an Interaction can happen, but we just have to wait and see whether or not the rest of it will happen as described. If the behavior exists exactly when what it represents is happening, then whether it exists at a time when part of it has happened depends on what will happen in the future. That is clearly nonsense. If we can make any sense of Interactions as a kind of Behavior, then it is a very different kind of behavior to the behavior of a classifier. The behavior instance can only come into existence after the described behavior has happened.

Reclassifying Interactions as a kind of Event would still not allow us to treat

---

2The situation may be compared to that of Schrödinger’s cat, which is meant to be both alive and dead.
Figure 5.10: Generalisation hierarchies of Interaction and the OccurrenceSpecifications
them as temporal logic formulae. This is because events, like behaviors, are also individuals that exist in the run-time system state.

The “Overview” Section of Chapter 13 of the definition describes the dynamic aspects of UML’s run-time semantics. To do so it employs a series of “domain models”, which it is careful to point out “are not metamodels but show objects in the semantic domain and relationships between these objects” [Obj07c §13.1]. That is, these run-time domain models are models at the same level as ordinary UML models, but they are universal, in the sense that they model every possible UML run-time system state. The domain models are “informal”, that is, non-normative, but as we show in Section §8.1 and Figure 8.4 (on Page 169), they could easily play an important role in making the run-time semantics precise. In Chapter 3 we developed an account of system state instantiation of a class model as graph homomorphism. The domain models could be incorporated by demanding that a modelled system state be homomorphically mapped into a combination of the user model and the semantic domain model.

In any case, the run-time domain model shows us what kinds of things we can find in a run-time system-state. An instance of an Event is called an occurrence [Obj07c §6.4], and Figure 13.2 in the definition shows a class EventOccurrence as part of a domain model. It is clear that these occurrences are not merely derivative of the difference between a system state and its predecessor, but are actual persistent entities in their own right: “When an event occurrence is recognized by an object, it may have an immediate effect or the event may be saved in an event pool and have a later effect when it is matched by a trigger specified for a behavior.” [Obj07c §13.1] There is further talk of occurrences being saved in “event pools”, “consumed” and so on, in [Obj07c BehavioredClassifier, §13.3.4; AcceptEventAction, §11.3.2].

Events and their occurrences must be distinguished from the related ideas Signal [Obj07c §13.3.24] and Message [Obj07c §14.3.20]. Signal is a kind of Classifier, and the instances of a Signal are units of asynchronous communication between objects. The generalisation hierarchy of Signal can be seen in Figure 5.11 (on Page 91). It would be more natural to call these things “messages”, however Message is reserved for another related concept. “A Message defines a particular communication between Lifelines of an Interaction.” [Obj07c §14.3.20]. The generalisation hierarchies of Message, and another important sequence diagram meta-class Lifeline, are shown in Figure 5.12 (on Page 91). The kinds of communication that the Message can represent include a Signal instance, a (synchronous or asynchronous) operation call, and creating or destroying an instance. On the face of it, Signal instances and operation calls are entirely different kinds of things. Signal is a classifier, so its instances are like objects, but an operation call is an action. However, the definition of that action makes it clear that, again a persistent entity is involved “CallOperationAction is an action that transmits an operation call request to the target object” [Obj07c §11.3.10]. The “domain model” shown in Figure 13.3] has a class Request. “Several kinds of requests exist between instances, for example, sending a signal or invoking an operation.” [Obj07c §13.1].
Thus Message is like an instance specification for Requests. The domain model shows that each Request has a sending and receiving object, as well as sending and receiving event occurrences.

Let us now summarise what we know about Events, Signals and Messages. Signals are like Classes. We use them to model the kinds of asynchronous messages that can be sent between objects in the target system. Their instances, like objects, have name-value pairs which constitute the content of the message. We can define kinds of signal by adding Signal elements to our model. Although it is not shown in the metamodel, Signal is “generalised” by Request, which also classifies the persistent entities which realise operation calls. Request instances can be specified by a Message. Events on the other hand are part of UML’s run-time machinery. We can not define new kinds of event, the available kinds are specified by the UML metamodel.

This “run-time machinery” defines the behavior of the systems which UML models describe. It must take responsibility for creating the event occurrences. That is, whenever something happens which an event occurrence can represent, the appropriate event occurrences must be added to the system state. This seems unproblematic for events representing sending or receipt of signals or operation calls, creation and destruction of objects, or the beginning or end of a behavior execution. The rules which define these system state changes must also specify the creation of the corresponding event occurrences.

---

3 New event types could be defined using a profile [Obj07c, §18], but this amounts to defining a new language based on UML.
It is not so obvious how TimeEvent and ChangeEvent \cite{Obj07c} \S13.3.27, \S13.3.7 occurrences should be created. These events are different in two ways from those just discussed. Firstly, they are not directed toward any particular object. If object a calls an operation of object b, then the InvocationOccurrence has a as its sender, and the ReceiveOccurrence has b as its receiver \cite{Obj07c} Figure 13.3. A ChangeEvent has a boolean expression, and the event “occurs” when the value of the expression goes from false to true (it does not occur when the value goes from true to false). The occurrence is not related to any particular object. Similarly, a TimeEvent has a time expression, and the event “occurs” at that time. CreationEvent and DestructionEvent \cite{Obj07c} \S14.3.6,14.3.7 occurrences are also presumably not directed toward any particular object, or at least in the case of DestructionEvent, not for very long! We shall return to this point shortly in Section 5.4. The second way in which TimeEvent and ChangeEvent occurrences differ from the object specific ones is that there will be infinitely many of them, even when there are only finitely many objects in the system state. There are obviously infinitely many points in time, and infinitely many boolean expressions change from false to true with every change in system-state.

This is the way UML is defined, but there is an equivalent formulation that has the advantages of being finite and more uniform. For these object-independent events, it seems more natural for the run-time machinery to be “aware” of the Event elements in the model, and to create the occurrences only when they are relevant. Where such occurrences are relevant, they are relevant only to certain objects, so, they can be “directed” toward them (by some appropriately labelled edge), thus achieving a uniform treatment of occurrences. This relevance principle could be extended to the other events. Although probably not helpful for our theory, this would be a useful optimisation for an implementation of the UML run-time.

We have seen that UML’s dynamics adds a lot of new kinds of entity to the system states. Things that we would have expected to be transient, like events and operation calls are actually first-class inhabitants of the system state. We have also seen some difficulties with the formulation of UML’s dynamics. Emergent behaviour should not be represented by persistent entities, because its not clear when they exist. Parts of the definition assume that all event occurrences are directed at particular objects, but this fails to account for time and change and destruction events. Despite these problems, the overall picture seems reasonably clear. We are now ready to study the diagrams and model elements concerned with dynamics.

5.4 Emergent Behavior

The sequence diagram of our example model, shown in in Figure 5.3 (on Page 81), represents an Interaction \cite{Obj07c} \S14.3.13. This is a kind of emergent Behavior. Although we found difficulties with this UML notion in the previous section, we will persist with it here to see to what extent we can make sense of it, and what might be done to improve it. Since the event occurrences referred to by the se-
sequence diagram are actually there in the system state, we do not need the semantic apparatus of temporal logic to understand them. In fact, we can treat them in much the same way as an object diagram: as a graph which must map homomorphically into the system state.

Figures 5.13 (on Page 94) and 5.14 (on Page 95) show most of the model elements represented by the example sequence diagram of Figure 5.3 (on Page 81). We have not shown the GeneralOrderings, which say in what order the occurrences must happen [Obj07c, §14.3.12]. The main units of meaning of such an Interaction are the OccurrenceSpecifications. These are qualified by an Event and a Lifeline. The MessageOccurrenceSpecifications are also qualified by a Message, and the ExecutionOccurrenceSpecifications by a BehaviorExecutionSpecification.

The Event represents a collection of related occurrences. SendSignalEvents and ReceiveSignalEvents each refer to a signal, and are thus fairly specific about the occurrences they represent. ExecutionEvents [Obj07c, §14.3.8] on the other hand have no means to specify whether it is the beginning or the end of a behavior execution that they represent, nor what behavior the execution instantiates. That is, every ExecutionEvent represents the beginning and end occurrences of every behavior execution.

Our Interaction uses the ExecutionEvents and ExecutionOccurrenceSpecifications in the context of a BehaviorExecutionSpecification, which links to start and finish ExecutionOccurrenceSpecifications. Message similarly distinguishes between sendEvent and receiveEvent, but there are also distinct Event metaclasses for send and receive events. One could construct an Interaction in which the sendEvent of a given Message actually specifies a receipt occurrence. The resulting Interaction would presumably be unsatisfiable.

Let us now consider what this interaction should look like as a graph. Similarly to class and object diagrams (Chapter 3), we hope to treat the Interaction by applying an interpretation function to the system state which will pick out the relevant model elements and produce a graph which is satisfied if it maps homomorphically into that system state. What follows is an investigation into what this interaction interpretation function ought to do, and what the historical parts of the system state graph ought to look like.

Events classify occurrences in much the same way classes classify objects. There may be occurrences which are classified by no Event, for example, a model might not have any Events in it, but there will still be occurrences in its system states. We have already established that an object can instantiate more than one class. Since the ReclassifyObjectAction [Obj07c, §11.3.39] association newClassifier has lower bound 0, it is also possible for an object to instantiate no class. It therefore seems appropriate to use instantiation edges in the graph form of the system state to show which occurrences are represented by which Events.

If Events are analogous to Classes, then OccurrenceSpecifications are analogous to InstanceSpecifications. They represent a single event occurrence which instantiates its associated Event. We can achieve this in a system state graph by turning the event links in the model elements object diagrams of Figures 5.13 and
Figure 5.13: Lifelines and messages of the sequence diagram (an Interaction owns most of these elements, but is not shown)
“A lifeline represents an individual participant in the Interaction” [Obj07c §14.3.19], where participants are objects or other Classifier instances. The Message has two ends, which are both OccurrenceSpecifications, the sendEvent and the receiveEvent. When the OccurrenceSpecification is at the sendEvent end of the Message, its Lifeline represents the sender of the Message, and thus the sender of the invocationOccurrence, as shown in [Obj07c Figure 13.3, Communication Domain Model]. Thus the covered link in the model of Figure 5.13 becomes a sender edge in the graph. The Message will represent a Request which is either a Signal or an operation Call. The label sendEvent on this link in the model is retained in the corresponding graph edge, because it matches the corresponding label on the system-state domain model.

The GeneralOrdering elements of the Interaction (not shown in our diagrams) are analogous to Properties, in that they will become edges under the interpretation function. The event occurrences in the system state will need to be partially ordered, chronologically. Since the system state is a graph, the natural way to represent this partial ordering is to have it generated by specially labelled edges between the occurrence nodes. The edges obtained from the GeneralOrderings will map to these partial ordering edges, or their composites. In this way, we obtain a graph which will map homomorphically into the system state if and only if the story described by the sequence diagram has happened in the specified order. We will ignore the GeneralOrderings, and sequencing edges, because they would add clutter and do not affect the satisfiability of our example sequence diagram.

The definition talks about Interactions representing “traces”, by which it means tuples of occurrences [Obj07c §14.3.13]. These tuples are the total orderings compatible with the partial ordering given by the GeneralOrderings in the Interaction. The occurrences in the system state need not be totally ordered, since it is possible for two things to happen simultaneously. Furthermore, simultaneous occurrences could be part of the same Interaction instance. Considering the total orderings of this inherently partial ordering just adds unnecessary complexity. Indeed, the meaning of an Interaction is given in terms of a pair of sets, a set of valid traces
and a set of invalid ones. This is to cater for a negation-like operator \[\text{Obj07c} \ §14.3.3 \text{ and } 16\]. Harel and Maoz argue that this semantics is “inadequate, and prevents the new operators from being used effectively” \[\text{HM07}\].

Figure 5.15 (on Page 97) gives an indication of the graph the interpretation function will obtain from the Interaction and its associated elements. This can be seen as the story of the sequence diagram, as it will appear in the recorded system history in the system state.

Interactions can be combined using CombinedFragment \[\text{Obj07c} \ §14.3.3\], which provides operations such as alternation, sequencing and repetition. The simple graph homomorphism approach outlined here would not be adequate to define the semantics of these operations. Instead, given for example, an alternation of Interactions, we should say that the combined Interaction was satisfied if graph \(g\) maps homomorphically into the system state, or graph \(g'\) one does, where \(g\) and \(g'\) are the graphs obtained from the two alternatives. These operations are already present in dynamic logic, giving us yet another reason to consider it as a means to define their semantics.

We conclude this Section by returning to an issue mentioned in Section 5.3, which on more careful examination seems to be a bug in UML.

The definition does not specify how long event occurrences persist. Do they cease to exist when removed from the event pool of an object? Even when removed from the last pool, the event occurrences are needed to satisfy Interactions. Since any Interaction can be extended with another OccurrenceSpecification, there is no limit to how long a given event occurrence might be needed. Again, some theory of “relevant” occurrences could be developed to support an equivalent but more efficiently implementable run-time. Perhaps we can cull the occurrences once they can no longer participate in an unsatisfied emergent behavior. This seems unlikely to admit a sufficiently simple and precise characterisation for inclusion in a semantic definition. We therefore assume that event occurrences persist forever.

This still leaves a serious problem. These immortal occurrences will suffer from a kind of “event-rot” as the objects they refer to are destroyed. This will inevitably lead to violations of the multiplicities of the run-time domain model \[\text{Obj07c} \ §13.1\], but this is only a minor issue. The real problem is most immediate in Interactions like the one shown in Figure 5.16 (on Page 98), which includes a Lifeline with some MessageOccurrenceSpecifications that ends with an OccurrenceSpecification of a DestructionEvent. If that destruction occurrence is not the last one specified in the Interaction, we have trouble. When the described happenings are complete, the send and receive event occurrences for the object represented by that lifeline must be connected to that object to show that it has played the specified role. But the destruction event ensures that the object is not there. Contradiction. The Interaction can never be satisfied.

This is not a bizarre special case, but a common usage for sequence diagrams. Indeed, a sequence diagram containing this pattern is given as an example in the official definition \[\text{Obj07c} \ Figure 14.11\]. A minor consolation is that it does not affect the language fragment required for our example model. The problem seems
Figure 5.15: The graph obtained from the sequence diagram model elements by the Interaction interpretation function
to be a result of the “reification” of event occurrences. To make sense of that, perhaps it is also required to “reify” existence itself, by introducing an existence node. Objects which exist would have an edge connecting them to existence, and non-existent objects would not. Perhaps temporal logic is a better way to handle descriptive dynamics after all?

5.5 Causation and Behavior

Having clarified how event occurrences come into existence, at least within the system boundary, we turn now to how they cause their effects. All event effects are effects on the behavior of an object. There are three kinds of model element that define the effect that a given event can have: Reception, Operation and Trigger. Receptions say what to do about receiving a signal, Operations say what to do about receiving an operation call. It’s not the signal or operation call itself that causes things, but rather the event occurrence that represents its receipt by the targeted object. Operations and Receptions are kinds of BehavioralFeature, as shown in the generalisation hierarchy shown in Figure 5.17 (on Page 99). Each BehavioralFeature has zero or more Behaviors associated with it, called its methods. These are the possible effects defined by the Reception or Operation. Triggers are more general than Receptions and Operations, in that they can define responses to any kind of event. On the other hand, the effect defined by a Trigger is not so explicit, and in some contexts it is completely unclear, as we shall see below. Trigger’s generalisation hierarchy is shown in Figure 5.18 (on Page 99).

There are two kinds of effect: immediate and triggered. Neither Receptions nor Triggers can define immediate effects, so the only kind of event that can have immediate effects are operation call receipts. The relevant events are ReceiveOper-

---

Footnote 4: Causation is not clearly explained in any one place in the definition, and the different passages are sometimes difficult to reconcile. It has therefore not always been possible to cite specific places to support specific assertions. We explain here our understanding as drawn from §6.3.3 The Basic Causality Model, §12.3.2 AcceptEventAction, §13.1 Common Behaviors Overview, §13.3.4 BehavioralClassifier, §13.3.22 Operation, §13.3.23 Reception, §13.3.24 Signal, §13.3.31 Trigger and §15.3.12StateMachine.
Figure 5.17: Generalisation hierarchies of Operation and Reception

Figure 5.18: Generalisation hierarchy of Trigger
ationEvent and CallEvent [Obj07c §14.3.27,§13.3.6]. These two Event types seem to be equivalent to one another, and also to the classes CallBehaviorOccurrence and CallOccurrence of the “domain model” [Obj07c Figures 13.2, 13.4]. The effect of an operation call can only be immediate if the object has no Trigger for the relevant event occurrence [Obj07c §13.3.22]. The effect is always invocation of a Behavior. An Operation is a BehavioralFeature of a Classifier. It can be associated with Behaviors (only in the §13.3.4 version, not the §7.3.5 one). How a behavior is chosen when the operation is called is a “semantic variation point”, to allow for different kinds of late binding.

When an event occurrence arrives at an object and there is no immediate effect, it is placed in the object’s “event pool”. As we discussed above, TimeEvents and ChangeEvents are not directed toward any specific object, yet have triggered effects. Its not clear whether these are placed in the pool of all existing objects, or whether the run-time mechanism is supposed to deliver them only where they are relevant or exactly what is supposed to happen.

We are told that “operations specify immediate or triggered effects” [Obj07c §13.3.22], but there is no way of specifying that a given operation has a triggered effect. Perhaps it means that operations have immediate or triggered effects? The operation could decline to respond to the event immediately, allowing it to go into the object’s event pool, and later handle it as a triggered effect. This conflicts with the assertion that “a triggered effect is manifested by the storage of the occurrence in the input event pool of the object and the later consumption of the occurrence by the execution of an ongoing behavior” [Obj07c §13.3.4] (emphasis ours). Another part of the definition describes the situation differently, making triggered operation invocation a possibility: “[the event occurrence] is taken from the input pool and either directly causes the occurrence of a behavior or [it is] delivered to the classifier behavior of the receiving object for processing” [Obj07c §13.3.31]. Some clarification of the definition is needed: can a behaviour be invoked as a triggered effect, or not?

Where Operations handle calls, Receptions handle Signals. Like Operations, Receptions are also a kind of BehavioralFeature. They can only produce “triggered” effects, since the notion of immediate effect is reserved for operations. Signals are always asynchronous, but the triggered/immediate distinction is independent of the synchronous/asynchronous one [Obj07c §11.3.1]. There seems to be no principled reason why receipt of an asynchronous signal should not have an “immediate” effect, but that is how the language is defined.

So, Triggers and Receptions can only specify triggered effects. When one of these things handles an event occurrence, it is removed from the object’s event pool, but as we discussed before, it continues to exist. If the object has no way of handling the event occurrence, what happens to it is a semantic variation point [Obj07c §13.3.4].

A Trigger can be part of a BehavioredClassifier, a Transition or an AcceptEventAction. Trigger only has NamedElement as a superclass, as shown in Figure [5.18](on Page 99). Each Trigger is associated with one Event, which specifies the kind
of event occurrence it is looking for. “A trigger specifies an event that may cause
the execution of an associated behavior” [Obj07c §13.3.31]. That is, a Behavior
associated with the BehavioredClassifier or Transition which owns the Trigger. A
BehavioredClassifier can own many Behaviors, so its not clear how the appropriate
behavior is selected. Although the definition does not say so, we presume it is a se-
matic variation point as we just discussed regarding the methods of an Operation
or Reception. An AcceptEventAction is already within an Activity, which must
be executing for it to accept an occurrence. They can be useful when the Activity
needs to receive a Request (signal or operation call) while it is running, and make
its contents available to the executing process.

A brief aside: there seems to be an anomaly with this action. AcceptEventAc-
tion “cannot be used with synchronous calls” [Obj07c §11.3.2, Semantics], and
its specialisation AcceptCallAction [Obj07c §11.3.1] cannot be used with asyn-
chronous calls. But the only way of specifying what event occurrences they accept
is the Event linked to its Trigger, and none of the kinds of Event for specifying oper-
ation call event occurrences have any way of distinguishing between synchronous
and asynchronous calls. Perhaps the definition is suggesting that the run time ma-
chinery will navigate from an event occurrence to the CallAction [Obj07c §11.3.8]
which originated it, and determine whether or not the call is synchronous, before
accepting or ignoring the call?

That is how causation works in UML. As we said earlier, every effect is an
effect on an object’s behavior, either initiating a behavior execution, or altering the
course of an already executing behavior. In Section 5.3, we mentioned that Interac-
tions are a kind of behavior, an emergent behavior. The kinds of Behavior that can
be executed by an active object are Activities [Obj07c Chapter 12] and StateMa-
chines [Obj07c Chapter 15]. Our small example actually has both of these: class
A has a StateMachine, and the entry action of its state s’ is an Activity. We discuss
the state machine in the remainder of this section, and in the next, we will examine
the entry action Activity.

In our example, the state diagram shown in Figure 5.2 (on Page 81) is intended to define the behavior of class A in the class diagram of Figure 5.1 (on Page 81). Class is a kind of BehavioredClassifier which has an association classifierBehavior to its Behavior. The metamodel elements of the state machine in Figure 5.21 (on Page 104) shows this link between A and its state machine. Whilst the ownedBehaviors of a classifier are there to be invoked in various situations, executing its classifierBehavior is part of what makes an object an instance of its class. “When an instance of a behaviored classifier is created, its classifier behavior is invoked” [Obj07c 13.2.4].

Figures 5.19 (on Page 103) and 5.20 (on Page 103) show the generalisation hierarchies of StateMachines and their main components. States and Transitions are familiar concepts, similar to their origins in finite state automata, especially when there are no “compound” states. One state is active, and when the Event on an outgoing Transition of the active state matches an event occurrence taken from the object’s pool, the state at the other end becomes the active one. There can be

101
entry and exit actions on the states, and actions on the transitions.

Regions allow state machines, and their states to be divided into concurrently executed parts. Our state machine has no concurrency, as it has only one region, and neither state contains a region. Generally, the “active” states of a state machine are a set conforming to the following rules: the state machine itself is active, each region immediately contained by an active state or state machine is active, and up to one of the states contained by an active region is active. The state machine itself can be seen as a tree with states and regions as nodes, alternating as we go deeper down the tree. The possible sets of active states of the state machine are the subtrees that are “full” at the state nodes, and have no more than one child at each region node. Such a subtree is uniquely determined by its leaves, so this would be the way to represent it in a graph: active edges from the behavior execution node to the active leaf nodes of the state machine. In our example, either \( s \) or \( s' \) is active, or that neither state is active.

State machine regions can contain an initial “pseudo-state” with a single outgoing transition that has no trigger \[^{[Obj07c\textsection15.3.8]}\]. Thus the real initial state of the region is the target state of that transition. Our example has no initial pseudo-state, and we have not been able to find a statement in the definition about what will happen when an execution of such a state machine begins. Perhaps it will have no active state, and thus never do anything. Also, it is a mistake to assume that every item in the system state is there as a result of a UML defined creation process. Although object oriented software systems work this way, UML also models the context of such a system. So for example an Actor may have a state machine describing its behavior, but we have no right to make any assumptions about what state we might find it in. Since UML is now applied beyond software, the same argument applies to all model elements, as we saw in Section 4.2.

Returning to the model elements of our state machine in Figure 5.21 (on Page 104), the Trigger of the Transition is connected to the same ReceiveSignalEvent which forms part of the Interaction (sequence diagram), which is in turn, connected to the Signal \( w \). It is essential that we reuse this Signal rather than creating a new one, because signal instances will generally instantiate only one Signal, the one associated with the SendSignalAction that created it. Even if we had two Signals named “\( w \)”, they would generally have different instances. (As we discussed before, it is possible for run-time entities to instantiate none or many model elements, for example when a ReclassifyObjectAction is applied.)

It seems unclear whether or not the Reception shown in Figure 5.22 (on Page 105) is needed. Reception’s semantics \[^{[Obj07c\textsection13.3.23]}\] say that “the receipt of a signal instance by the instance of the classifier owning a matching reception will cause the asynchronous invocation of the behavior specified as the method of the reception” (emphasis ours). We do not want to invoke the behavior, we want the occurrence to be processed by the behavior execution that is already running. However, the description of StateMachine \[^{[Obj07c\textsection15.3.12]}\] says that “The behavior classifier context owning a state machine defines which signal and call triggers are defined for the state machine, . . . signal triggers and call triggers for
the state machine are defined according to the receptions and operations of this classifier.” The semantics of BehavioredClassifier [Obj07c §13.3.4] does not give the impression that anything extra is needed for behavior executions to find the occurrences in the pool of their object, nor does the semantics of Trigger [Obj07c §13.3.1], which says that “an event is dispatched when it is taken from the input pool and either directly causes the occurrence of a behavior or are delivered to the classifier behavior of the receiving object for processing”. The meaning that the definition gives to our state machine is now as clear as we can make it, except for the entry action of state \( s' \). This will be the subject of our next section. We have found a few more problems with the existing definition. It is unclear whether or not operation invocation can be a triggered effect, nor is it clear whether or not a Reception is needed to transfer event occurrences to a StateMachines Trigger. Different parts of the definition take different points of view.
Figure 5.21: Metamodel elements of the state machine diagram
Figure 5.22: The definition appears to contradict itself on whether or not a Reception is needed to join class A and its state machine

5.6 Graph Transformation and Action

Our working hypothesis has been that UML system states are graphs. The question now arises, how do we get from one system state to another? As we shall see in Section 7.2, many authors have suggested graph transformation systems as a formal semantics for the dynamic aspects of UML. However, none of these authors have accounted for the actual details of UML’s dynamics as currently defined. This is mostly because many of the details are new with version 2.0, which was first officially released in July 2005 [Gro08]. In this section, we will see that graph transformation really is a good match for the dynamic ideas UML embodies. As usual, our investigation is guided by the relevant fragment of our example model, in this case the entry action of the state $s'$ of the state machine (Figure 5.2 on Page 81).

The different kinds of Behavior, such as Activity andStateMachine are essentially ways of structuring collections of Actions. It is the Actions which really do things, make things different in the system state that follows their execution. Action is the fundamental unit of change in UML. The only things that happen in a UML defined system are objects performing actions, and objects accepting event occurrences, as discussed in Section 5.5. Each of these units of change are well suited to being defined by a graph transformation rule.

Graph Transformation [BH02, EEPT06] is the graphical equivalent of term rewriting [BN98], a standard way of defining operational semantics. We will examine graph transformation and the UML related graph transformation literature in Section 7.2. For now, it is enough to note that we can define an evolution relation over a space of graphs by providing rules which consist of “before” and “after” subgraphs. Whenever a system state contains a copy of a “before” part of a rule,
we can replace it with the “after” part, and obtain a valid successor state. As we examine the metamodel elements of our example action, we will consider what kind of graph transformation rules will be required to realise it.

We turn now to the specific example before us, and the meaning of the model elements it comprises. The generalisation hierarchy of Activity is shown in Figure 5.23 (on Page 106). The metamodel elements of the entry action of state $s'$ of our example are shown in Figure 5.24 (on Page 107).

A behavior execution in the system state will need to record its progress in much the way the instruction pointer register of a CPU does, or the program counter of a virtual address space. The control structures are described in [Obj07c, Chapter 12], and are defined in terms of “tokens” being passed from one ActivityNode to another. Actions also have data requirements. When the control and data requirements for an action are satisfied it is “enabled” and may be executed. The obvious way to do these things in a system state graph is to have one or more specially named edges from the execution to the next Action or Actions to perform. The graph transformation rules which perform the actions will also be responsible for updating these “enabled” edges.

The flow of data between the Actions in a Behavior is mediated by Pins [Obj07c §11.3.28, §12.3.44]. Pins are temporary storage areas where one action can leave some values for another action to use. They have a type, and a multiplicity, as the specialisation hierarchy shows in Figure 5.25 (on Page 108). Actions wait until the required number of values are available on their input Pins. This required number is given by the lower bound. Since all the upper and lower bounds of pins in our example are 1, we have not shown the upper and lower bound objects in Figure 5.24 (on Page 107) as we did in the class diagram model elements diagram 5.8 (on Page 85).

Recall the discussion in Chapter 3 of the core metamodel fragment shown in Figure 3.2 (on Page 36). This fragment has classes Property and Classifier and two associations between them, classifier and type. We noted that this fragment resembles the category theoretic diagram which defines graphs. We have an analogous situation here with Pins and Behaviors. Because Pin is a kind of TypedElement, each Pin has a type which is a Classifier. Each Pin also belongs, perhaps transitively, to an Activity, which is a kind of Behavior and hence a Classifier. In Chapter
Figure 5.24: Metamodel elements represented by the entry action of state s’
we converted the model graph with properties as nodes into a type graph with properties as edges. The equivalent move here is to turn the Pins into labelled edges, so that valid run-time states will have edges from behavior executions to their pin-values, labelled by the pins name. We shall adopt this as a working hypothesis of run-time pin values.

A difficulty with this proposal is that, as a Classifier, Behaviors can have Properties, and a Property might have the same name as one of the Behaviors Pins. If a behavior execution has an outgoing edge \( p \) to some value \( v \), there will be no way of knowing whether \( v \) is on the pin \( p \) or the property \( p \). This is not much worse than a difficulty already present in the graph treatment. A classifier can have two or more Properties with the same name. An outgoing edge labelled with this name might belong to either of them. A ReadStructuralFeatureAction associated with one of these Properties would return values that are meant to belong to the other. One option would be to label the edges with the Property or Pin itself, i.e., the value which is the node in the model graph?

Let us consider the Actions that make up the example Activity. Recall that it can be expressed textually as \( \text{send X to self.ex} \). The “innermost” part of this command, the part that must be evaluated first, is \( \text{self} \), which is represented in the Activity as a ReadSelfAction [Obj07c §11.3.36]. Assuming that all pin instances are empty when a behavior execution begins (the definition does not say), this is the only Action which can be executed immediately, as it is the only one with no input pin. “When the behavior executes, it does so in the context of some specific host instance of that classifier. This action [ReadSelfAction] produces this host instance, if any, on its output pin” [Obj07c §11.3.36].

An ActionInputPin takes the values on the output pin of an action and provides them as input to another action. If we realise values being on pins by edges labelled with the pin name, then an ActionInputPin should cause these edges to be created in pairs. Whenever a value is added to the output pin of the fromAction of an ActionInputPin, it should also be added to the ActionInputPin. In our example, when the context object \( a \) is put on the output pin of the ReadSelfAction by adding a “\( \text{readSelfOPin} \)” labelled edge from the behavior execution to \( a \), a “\( \text{readSelfIPin} \)” labelled edge should also be added there.

Once this has happened, the ReadStructuralFeatureAction will be enabled. The
other parameter of this action is the Property “ex”, which we saw before in the model elements from the class diagram, in Figure 5.8 (on Page 86). We discussed and defined ReadStructuralFeatureAction in Chapter 4. There, we determined that each Property in a reflective system state denotes a function from the instances of its classifier to a collection of values of its type. The ReadStructuralFeatureAction evaluates this function for its Property and the instance on its input pin, and places the resulting value on its output pin. The multiplicities of the property ex and of pin are both 1, so in this example there is no problem about the number of values on the pin. But what happens if the number of results is more than the upper bound of the output pin? “An action may not put more values in an output pin in a single execution than the upper multiplicity of the pin” [Obj07c, §11.3.27]. Perhaps an arbitrary selection is made, or it is considered to be a run-time error?

The input parameters to the SendSignalAction then are the (one and only) ex of the object executing this action, which it will find on the pin, and the Signal x. This Signal does not occur in the Interaction model elements in Figure 5.13 (on Page 92). Execution of the SendSignalAction will result in a new instance of Signal x being created. According to our interpretation of the run-time “domain model” of [Obj07c, Figure 13.3], it will have an outgoing edge labelled “sender” to the object a, and one labelled “receiver” to the object b. A separate rule will add the receive event occurrence and its edges, message to the x signal instance, and a pool edge into it from b. This creates the possibility of a delay between sending and receipt of a signal, as required by the UML run-time definition.

We now have a fairly clear idea of the system states that our example model will have during its possible executions. Graph transformation rules seem a good candidate for a way of defining this precisely. Another example of the graph-defining graph has emerged with Behaviors and their Pins. We now turn to the task of this Chapter, to determine whether or not the model is consistent.

5.7 Consistency

The previous sections of this chapter have studied the individual model elements of our example model. Now it is time to put what we have learned together, and ask whether the model is consistent. We shall first establish just what that means, and uncover some more difficulties overlooked in the official definition.

In Section 5.4 we determined that an Interaction (sequence diagram) is satisfied by a system state in much the same way as an object diagram is. Because the system state contains its own history, we do not need to consider a sequence of system states, just one state from after when the described happenings are complete, but before any of the participants have been deleted. As we observed, this is not possible for some sequence diagrams containing DestructioEvents.

The graph in Figure 5.15 (on Page 97), which we obtained from the Interaction and the elements linked to it, is a picture of the history we expect to find in the system state. In Section 3.6 when considering the semantics of InstanceSpecifications
and Slots (object diagrams), we determined that the homomorphism that maps the graph obtained from these elements into the system state must map any Classifiers to themselves. Otherwise, the InstanceSpecifications might refer to objects of the wrong Class. The same constraint must apply to the interaction graph.

The “initial” state of a system could contain objects, links and behavior executions, but should not contain event occurrences. Allowing this would be equivalent to the implausible situation described by Bertrand Russell in response to creationist claims that God created the earth complete with buried dinosaur skeletons: “There is no logical impossibility in the hypothesis that the world sprang into being five minutes ago, exactly as it then was, with a population that “remembered” a wholly unreal past” [Rus21]. In the case of our example model, allowing “pre-existing” event occurrences would make its consistency trivial, we would only need to construct a system state that contains the graph shown in Figure 5.15. Whether or not the described events can actually happen would be beside the point. We therefore further restrict the valid UML system states to those that either contain no event occurrences, or are derived from such a state by valid system evolution. That is, we prohibit the system state from having an unreal past.

What we are attempting to find then, is an initial state, and a sequence of system-state transitions which will result in a system state containing the interaction graph of Figure 5.15 as a subgraph, perhaps with some renaming.

The required initial state will have objects ee, a and b. Lifeline does not specify the Classifier(s) of the objects they represent, so they may or may not instantiate Classifiers EE, A and B respectively as we intend. In Section 3.5 we determined that the model must be present in each system state, so these Classifiers will be present, along with the send and receive signal event elements from the Interaction.

We have discussed the creation of event occurrences, but only where they originate from inside the system. But not all events are generated by objects within the system. “Events are often generated as a result of some action either within the system or in the environment surrounding the system.” [Obj07c, §13.3.31]. Unfortunately, this passage does not go on to give details. The Actor in our example sequence diagram is intended to represent someone or something outside the system boundary. Although she is outside the system boundary, we distinguish her from the remainder of the system environment. If we were content to treat the system context as an undifferentiated whole, we would dispense with the Actor. The signal could just appear as a “found message” [Obj07c, §14.3.20] which is received, but never sent. This would be compatible with Mellor’s original example, since he does not specify what caused the first object to emit its signal. We prefer to try to make sense of our formulation however, since Actors are a part of UML, and intended to be used in this way.

Actor is a kind of BehavioredClassifier, so it can have state-machines and activities. Indeed, it appears that our Actor must have a behavior, because it is intended to send a message. The send message event occurrence would instantiate

5This would violate the multiplicities of the system-state domain model [Obj07c, Figure 13.3]
InvocationOccurrence in the definitions system-state domain model [Obj07c, Figure 13.3], and Each of these is linked to exactly one behavior execution. How do we reconcile prescribing the behavior of an element with the fact that that element represents an entity that is not controlled by the system? One solution is to give the Actor an OpaqueBehavior. The semantics of OpaqueBehavior is “determined by the implementation” [Obj07c, §13.3.20], which is to say, it could do anything.

This would seem to make the model’s consistency a trivial question. If the behavior can do anything, then anything can happen, so the story told by the sequence diagram can happen, so the model is consistent. What we want is to describe a range of possible behaviors for the Actor. Because of the parallel, non-deterministic execution model of UML’s Activities, this may be possible. It is not obvious how to define the range of possible recipients of SendSignalActions though. We could solve that problem for our example by assuming that a link exists in the initial state between the Actor ee and the object a. This leaves the general problem of specifying a limited range of possible actions for Actors.

We assume that in the initial state, the external entity ee is already executing an opaque behavior eebeh. This behavior adds the Signal instance w to the system state, along with its send event occurrence wsoc. If this non-deterministic behavior can be properly defined in UML, then the graph transformation rules will follow, but if it needs to be an OpaqueBehavior, then some kind of wild-card rule will be required. The outgoing instantiation, sender, receiver, sendEvent edges will also be added at this point.

A separate rule will add the receive event occurrence wroc and its edges. This creates the possibility of a delay between sending and receipt of a signal, as required by the UML run-time definition. An edge labelled pool will connect a to this occurrence, indicating that it is in a’s event pool.

The presence of this occurrence in a’s event pool will enable a match with the left-hand-side of an event acceptance rule, which must also match many other aspects of the system state. There will need to be three versions of this rule, one for each of the possible effect defining elements: Reception, Operation and Trigger. Recall that there is some doubt about the meaning of Reception, and it remains unclear whether class A should have a Reception for Signal W. The Trigger form of the rule may need its own variants to deal with the different roles a Trigger can play. So, a Transition Trigger form of the accept event occurrence rule will match the object a and the event occurrence wroc and the pool edge between them. The active link from a to the origin state S of this Trigger will also occur in the left hand side of this rule. The Trigger is connected to the Signal, and the event occurrence is also, via its instantiation of wRecEv, the receive signal Event. The right hand side of the rule will omit the pool link, will have the active link moved from the source to the target state of the Transition. This rule will also need to create any new behavior executions for the exit action of the source state, the effect of the Transition, and the entry action of the target state. Indeed for nested states, the behavior invocation story is even more involved [Obj07c, §15.3.11, State].

After this rule has been applied, there will be a new execution of the Activity
which is the entry action of state $s'$. It will be hosted by $b$, and each of its actions will be processed by a graph transformation rule dedicated to that Action, maintaining pin value edges as described earlier. When it reaches the SendSignalAction, the resulting signal node will instantiate Signal $X$, not $Y$, because $X$ is the signal linked to that action.

In the early phases of development, UML models are not intended to describe every detail of their target system. Thus valid interpretations should be allowed to contain items not described by the model. If such an object received a $w$ message from $ee$ and sent a $Y$ message to $b$, then it can play the role of $a$ in the interaction, and it is thus satisfied. For that to happen, the object would have to instantiate a classifier with a behavior which would make it send the message. We tentatively defined the valid states of a given model to be those that contain the model, and have certain homomorphisms to and from various interpretations. This allows the model in the system state to be a superset of the one specified.

Although this relaxed view has its uses, perhaps in later stages of development, and in particular when checking sequence diagrams, we should be stricter about what is permitted in the system state. We could exclude non-model classes by insisting that $(P \circ \gamma)(ss) = (P \circ \gamma)(m)$, where $m$ is the model, $ss$ the system state and $(P \circ \gamma)$ is the class model interpretation function from Chapter 3. Let’s add this constraint to the admissible system states.

Now, because the map from the interaction graph to the system state graph must preserve classifiers, the edge $Y \xrightarrow{X} Y$ will have nothing to map to. There can not have been any $Y$ instances in the initial state, because each request must have a send and receive event occurrence [Obj07c, Figure 13.3], but we required that the initial state had no event occurrences. There is no classifier in the model that has a behavior which can produce a $Y$ signal.

We conclude then that, contrary to Stephen Mellor’s claim, the model is inconsistent, provided that the model is seen as complete, ie that the system contains no other classifiers. We have encountered much that is unclear and difficult in the UML definition, and our conclusion depends on some kind of resolution to these issues. We summarise them here for completeness.

1. There are many problems regarding “emergent behavior” such as Interactions:
   - Valid system states should only have real history recorded in them.
   - Many intuitively sensible interactions are unsatisfiable because of the “event rot” problem.
   - The definitions “trace” semantics for Interactions is an unnecessary complication. Our graph homomorphism approach is equivalent and much simpler. (However, the whole “recorded-history” approach to descriptive dynamics is flawed and probably should be replaced by a temporal logic approach.)
Emergent behaviors are really quite different to other Behaviors. It should not be a kind of classifier, because the status of partially completed executions is ambiguous.

2. Is a triggered effect always accomplished by an ongoing process consuming the event occurrence, or can operation invocation be a triggered effect?

3. Is a Reception needed for a classifier’s behaviors to handle its signals?

4. The partial and complete views of the model’s specification of the system should be addressed in the definition. This is a global semantic variation.

5. Event occurrences such as TimeEvents, which are not directed toward specific objects, need to have their effect’s process defined. Does every UML system state contain infinitely many such event occurrences?

6. The selection of a method for Receptions and Triggers owned by classifiers needs to be defined, or explicitly made a semantic variation point.

7. ExecutionEvents are extremely general. This could lead to unintended satisfaction of Interactions and Triggers.

8. There should be some observable distinction between synchronous and asynchronous call requests.

9. What happens when an action produces too many results for its output pin?

10. When a behavior is invoked, all the pins of the new behavior execution should be empty.

11. We require a way of defining a wide, but not unlimited range of possible behavior for Actors.

Although we have concluded that there really is a useful meaning to UML models, it is certainly not easy to determine this from the definition. There would be scope for other readers to disagree with the meaning we have extracted. Our graph hypothesis has been helpful to clarify things, and we do not believe that it has added meaning where there really is none.

In Chapter 11 we set ourselves the task of solving the UML “definition problem”. Perhaps the most important conclusion to draw from the present Chapter is that this task makes sense: there really is something to define, and it really is not very well defined at the moment.

Since the graph hypothesis has helped us to explain all we have encountered in UML, we propose that a definition be written directly in terms of graphs. Working software developers are notoriously averse to anything that looks mathematical. However, the required background is easily within reach of undergraduate discrete mathematics courses. It is no more difficult than the relational material covered
in any worthwhile university database course. We compare the state of database theory with that of UML in Section 7.1 and find that the relational model is an example UML should aspire to. The proposed definition document would need a few pages of introduction to graphs and graph transformations. The cost of each attempt to extract meaning from the current definition is quite unreasonable. We believe that the investment asked of a reader in reading the introduction to graphs would be returned many-fold each time she attempted to determine the meaning of a given modelling construct.
Chapter 6

Dynamic Logic for UML Consistency

In Chapter 5 we concluded that the current UML definition \cite{Obj07c} does define its semantics sufficiently to declare the tiny example model of Section 5.1 inconsistent. There were many provisos to this conclusion, and many ways in which the clarity of the definition needed improving. We also found several serious difficulties with a rather fundamental aspect of UML, its descriptive dynamics.

Each UML system state records its history as event occurrences which are linked to one another and the things they happened to. Interactions (sequence diagrams) describe these configurations of recorded history. In fact, they are defined as an “emergent behavior” and actually specialise Behavior and thus Classifier, and should therefore have instances. The best sense we could make of that is that the recorded history subgraph is the “instance”, which only exists after the described occurrences have happened. The most serious flaw in this scheme is that if some of the participants cease to exist, the recorded history can no match the description given by the Interaction. Therefore, Interactions that describe destruction events followed by something else are unsatisfiable. We called this the “event rot” bug.

We proposed that descriptive dynamics would be better handled by a temporal logic\footnote{By this we mean any modal logic whose modalities represent the passage of time, not only those logics with binary “until” operators.}. Although the task of this study is to define the existing language better, this Chapter explicitly exceeds that scope to consider modifications to UML itself.

The main part of this chapter is work we completed about \(2 \frac{1}{2}\) years ago, before we recognised the deep parallels between UML and graph transformation. Its goal was to make UML precise by translating its diagrams into dynamic logic \cite{HKT00}. It does not consider the official UML definition, rather it treats the fragment used in the Executable UML technique \cite{MB02}, and follows its conventions. The work was published as \cite{O'K06a}. Although classical dynamic logic for UML semantics no longer seems like a good idea, there is value we believe in its approach to descriptive dynamics. We present the work in close to its original form, then in
a concluding postscript section, consider whether descriptive dynamics would be better defined using this approach, and how it might fit in with the remainder of the language.

The first section briefly introduces dynamic logic. In the following section we systematically translate each diagram of the example model from Section 5.1 into dynamic logic formulae. A system specification is then formed using these diagram formulae as subterms. We then search for a trace which satisfies this specification and establish that none exists.

6.1 Dynamic Logic

In this section we briefly introduce logic, beginning with simple propositional logic. Then we consider two different extensions: modal logics and first order logic. Finally, we combine these extensions and obtain the form of dynamic logic we need to complete our formalisation.

A logic consists of syntax, semantics and a deductive calculus. The syntax defines a set of formulae, which we call the language of the logic. The formulae are just symbolised statements. The semantics defines a range of possible situations, each of which assigns either true or false to each formula of the language. If a situation $w$ makes a formula $\varphi$ true, we write $w \models \varphi$, and, confusingly for UML people, say that $w$ models $\varphi$. We say that $\varphi$ is a semantic consequence of a set of formulae $\Gamma$, if $\varphi$ is true in every situation that makes all the formulae in $\Gamma$ true. We also write $\Gamma \models \varphi$ when $\varphi$ is a semantic consequence of $\Gamma$. A deductive calculus defines proofs, each of which derives a formula from some set of formulae. If a proof exists deriving $\varphi$ from $\Gamma$ we write $\Gamma \vdash \varphi$. We want $\Gamma \models \varphi$ whenever $\Gamma \vdash \varphi$, so that deduction can be used to establish semantic consequence. We will not say much more about deduction.

Propositional logic has atomic formulae $P, Q, R, \ldots$ and if $\varphi$ and $\psi$ are formulae, then so are $\neg \varphi$, $\varphi \lor \psi$, $\varphi \land \psi$ and $\varphi \rightarrow \psi$. These symbols stand for not, or, and, and if . . . then . . . respectively. The possible situations of propositional semantics are functions from the atomic formulae to the truth values $\{true, false\}$. We will call these functions propositional interpretations. These are extended to the whole language by assigning each of the connectives $\neg$, $\lor$, $\land$, $\rightarrow$ the obvious truth function. We write $\top$ as a formula that is always true, and $\bot$ for one which is always false.

Propositional modal logics add some one-place connectives. Typically we add to the above syntactic rules that if $\varphi$ is a formula, then so are $\Box \varphi$ and $\Diamond \varphi$. Some intuitive interpretations of these connectives are: necessarily and possibly, always and sometimes, obligatory and permissible. We are interested in temporal interpretations, where $\Box \varphi$ means that $\varphi$ is true at all possible future situations, and $\Diamond \varphi$ is true in some possible future situation. Semantics for a propositional modal logic are given by introducing a binary relation $R$ between the propositional interpretations. Then $\Box \varphi$ is true at $w$ if $\varphi$ is true at every situation $R$-related to $w$, and $\Diamond \varphi$...
is true at $w$ if $\varphi$ is true at some situation $R$-related to $w$. So for example, we might think of the relation as representing time, so that $w_1Rw_2$ if our system can evolve from situation $w_1$ to situation $w_2$. Using the relation, the semantics are extended to formulae with $\Box$ and $\Diamond$ as follows: $w_1 \models \Diamond \varphi$ iff $w_2 \models \varphi$ for some $w_2$ where $w_1Rw_2$, and $w_2 \models \Box \varphi$ iff $w_3 \models \varphi$ for every $w_3$ where $w_1Rw_2$.

This is already a useful formal language, because if $R$ captures the possible evolution of a system, and we have formulae $\text{Init}$ and $\text{Bad}$ which represent the acceptable initial states of the system, and undesirable situations respectively, then the formula $\text{Init} \longrightarrow \Diamond \text{Bad}$ is true if and only if it is impossible for the system to evolve from an acceptable initial state into an undesirable situation.

Propositional dynamic logic PDL has a pair of modal operators for each program in a simple programming language. There are atomic programs, $\alpha, \beta, \gamma, \ldots$ and if $\rho$ and $\sigma$ are programs then so are $\rho; \sigma, \rho \cup \sigma$ and $\rho^*$. These are the regular expressions over the atomic programs. Also, if $\varphi$ is a formula, then $\varphi?$ is a program. Each atomic program denotes a relation over the situations, $\rho \cup \sigma$ denotes the union of the two relations (non-deterministic choice) and $\rho^*$ denotes the reflexive transitive closure of the relation denoted by $\rho$ (non-deterministic repetition). The program $\varphi?$ denotes the relation $\{(w, w) \mid w \models \varphi\}$ which relates a situation to itself when $\varphi$ is true there. This can be used to place guards on programs, and to write conditionals, such as $(\varphi?; \alpha) \cup (\neg \varphi?; \beta)$ for if $\varphi$ then $\alpha$ else $\beta$.

In propositional modal logic, the semantics for the modal operators $\Box$ and $\Diamond$ were given using a binary relation. In propositional dynamic logic, each program $\rho$ corresponds to a binary relation, and the semantics of the modal operators $[\rho]$ and $\langle \rho \rangle$ depend on this relation. Note the use of the angle and square brackets to resemble the diamond and square operators.

We can write for example $\langle \alpha \rangle \top$, to mean that the program $\alpha$ runs successfully (terminates), or $\langle \alpha \rangle \top \longrightarrow \varphi$ to mean that $\alpha$ only runs successfully in situations satisfying $\varphi$.

Although we work with full first order dynamic logic DL in this paper, it is likely that models could be usefully approximated by PDL. This could be enabled using automated tools, as PDL is decidable [HK10, Chapter 6], whereas DL, since it properly extends first order logic, is not.

First order logic also extends propositional logic. Where the basic formulae of propositional logic are unanalysed propositions $P, Q$ etc, first order logic formulae assert properties of individuals or assert relationships between individuals. For example, a two place relation symbol $L$ might be interpreted as “...loves...” the name $a$ might mean “Aaron” and $b$ “Belinda” then $\text{Lab}$ would be read as “Aaron loves Belinda.” The logic includes equality, so we may write $a = b$ meaning “Aaron is Belinda.” Names are one kind of term, that is expressions which refer to an individual. Variables $x, y, z, \ldots$ are another kind of term, and terms can be formed by applying $n$-place function symbols $f, g, \ldots$ to $n$ terms, for $n = 1, 2, \ldots$. For example if the 1-place function symbol $f$ is read as “the father of...” then $Lxf(b)$ should be read as “$x$ loves Belinda’s father.” First order logic also has quantifiers $\forall$ and $\exists$ so that $(\forall x)Lxf(b)$ means everybody loves Belinda’s father,
The semantic situations for first order logic (which logicians call “models”) consist of a set of individuals, called the semantic domain, and an interpretation which takes each name to an individual and each n-place relation/function symbol to a n-place relation/function. To evaluate variables and quantifiers, we also need a valuation. This takes each variable to an individual in the semantic domain. The formal definition of truth of a formula in an interpretation just says that, in the interpretation, things are as the formula says they are. The frightening notation required to state this precisely is unhelpful in the current context (horror enthusiasts are referred to [Hod97, §2.1]).

The semantics of quantified formulae are defined using the idea of variants of the valuation. An x variant of w is a valuation that is the same as w for all inputs except for x. This is worth explaining, because we will use these ideas again soon. We introduce some notation for a function the same as w, except that it takes x to q. Define \( w \oplus x \mapsto \rightarrow q \) by

\[
(w \oplus x \mapsto \rightarrow q)(y) = \begin{cases} 
q & \text{if } x = y \\
w(y) & \text{otherwise}
\end{cases}
\]

Then \( \mathcal{M}, w \models (\exists x)\varphi(x) \) iff \( \mathcal{M}, (w \oplus x \mapsto \rightarrow q) \models \varphi(x) \) for some q, and similarly for \( \forall \) formulae. Then \( \exists x, \varphi(x) \) is satisfied by the model and valuation \( \mathcal{M}, w \) iff \( \mathcal{M}, (w \oplus x \mapsto \rightarrow q) \) for some q. And similarly for \( \forall \) formulae.

The language of dynamic logic includes that of first order logic plus modal operators similar to those of PDL. The atomic programs of DL are assignments of the form \( x := t \) for some variable x and some term t. Programs are formed from atomic programs, using \( \cup, ;, \) and \( * \), and from formulae using \( ? \), just like in PDL, and the meanings of these connectives are also the same here. Thus to complete the semantics of DL we need to define the relation denoted by each assignment statement. The situations that make our formulae true or false are model, valuation pairs \( \mathcal{M}, w \). Intuitively, assigning a new value to a variable changes the valuation, but does not change the interpretation. So, for each interpretation \( \mathcal{M} \), the atomic program \( x := t \) relates each valuation \( w \) to \( w \oplus x \mapsto \rightarrow t^{\mathcal{M},w} \), where \( t^{\mathcal{M},w} \) is the value of the term t under the interpretation \( \mathcal{M} \) and valuation w. For example, the formula \( a = 5 \rightarrow (x := a)x = 5 \), says that if \( a = 5 \) then after you set x to a, you get \( x = 5 \). This is always true, because if \( a = 5 \) in \( \mathcal{M}, w \), that is \( a^{\mathcal{M},w} = 5 \), then \( x = 5 \) in \( \mathcal{M}, (w \oplus x \mapsto \rightarrow a^{\mathcal{M},w}) \).

Our objective is to reason about object oriented systems, where objects retain their identity over time, but have attributes whose values may change. So far, we have only seen how to make variables change over time. We could make the variables denote records, but this leads to object aliasing problems [CIO00]. If x and y denote the same object, then changes to x should result in changes to y, but without special tricks (eg [Bec01]), logics do not do this. A better way is to have object identifiers as individuals, and object attributes as functions, so that the familiar o.a notation becomes short-hand for \( a(o) \). Then what we want is the ability to update these functions. An extension of dynamic logic studied in [HKT00]...
allows such updateable functions, called array variables. Indeed, a similar system has been used to formalise parts of UML 1.1 in \([WB97]\) (discussed in Section 7.3).

Weiranga’s logic has ordered sorts too, to cater for specialisation.

The syntax of DL is extended by \(n\)-place array variables for each \(n = 1, 2, \ldots\). These can occur wherever an \(n\)-place function symbol can occur, but also on the left hand side of an assignment statement. We are mostly interested in 1-place array variables. We will use the attribute access dot notation when the array variable represents an attribute, so an assignment has the form \(t.h := s\) or \(h(t) := s\) for terms \(t, s\) and 1-place array variable \(h\). The semantics are adapted so that now the valuations also assign an \(n\)-place function to each \(n\)-place array variable. For a fixed model \(M\), this assignment denotes a relation that relates each valuation \(w\) to \(w \oplus h \mapsto (h^w \oplus t^M, w \mapsto s^M, w)\). That is, \(w\) is related to an updated form of \(w\), which maps the array variable \(h\) to the same function \(w\) maps it to, except updated so that it sends the value of \(t\) to the value of \(s\).

This is the form of dynamic logic that we use to precisely express our interpretations of the UML diagrams and actions.

### 6.2 The Model as Dynamic Logic Formulae

How can we represent this model using formulae of dynamic logic? We begin this section with some general considerations about the relationship between dynamic logic and the small UML subset used for this model. In each of the following subsections, we will discuss one of the diagrams, and give its meaning as a dynamic logic formula. These meanings are influenced by the use made of the diagrams in the Executable UML \([MB02]\) method. In the next section we will combine the class diagram, state machine diagram and action formulae to specify the system, and then show that this specification is inconsistent with the sequence diagram formula.

We give rather weak interpretations of each diagram, for example, assuming that there might be objects in the system that do not belong to any class on the class diagram. These interpretations are debatable, indeed it is possible that the weak interpretations are more appropriate in an analysis phase, whilst stronger ones might serve the design phase better. Indeed, we may want to keep several interpretations available to cater for UML’s numerous “semantic variation points.” The particular interpretations are not the main point though. Rather we aim to demonstrate that dynamic logic provides a simple and useful way of giving the meanings of the diagrams. The reader should not overlook another important virtue of the weak interpretations: they are shorter!

**System Snapshots and Evolution**

In Section 6.1 we said that the possible situations of the semantics of dynamic logic consist of an interpretation and a valuation. The interpretation gives meaning
to the fixed vocabulary such as names, relation symbols and function symbols. In the UML context, the functions and relations will mostly come from the OCL library, or perhaps an action language library. These can be considered to be fixed once and for all (though who knows what future versions of UML will bring?) If the interpretation was fixed throughout our work, that would be good, because we could essentially forget about it and only think about the valuations. For this reason, we will consider all model specific vocabulary to be variables, evaluated by the valuation, while the interpretation takes care of the global UML vocabulary such as OCL library functions. The valuation tells us about a particular system at a particular point in time. Hence, a system snapshot is a valuation, and the interpretation can be largely ignored, because it is fixed.

Each DL program relates pairs of these snapshots, but not every DL program corresponds to a legal evolution of the system. What we need is a formal definition of legal evolution. So, what can happen in a system defined by our subset of UML? Only two things really: objects can send messages, and they can accept them. There are conditions though, an object can only send a message if that action is at the head of its todo list2 although an external entity can send whatever it wants whenever it wants. An object can only accept a message if it has a message to accept, and it is not currently activated. The non-deterministic DL program ε which describes how these systems can evolve is defined as follows.

\[
\varepsilon \equiv ((sendCond(x, M, y)\?; x.send M \text{ to } y) \cup (acceptCond(x)\?; x.accept))^* \\
\]

where \(sendCond(x, M, y)\) is a formula, defined below, stating the conditions under which it is OK for \(x\) to send an \(M\) message to \(y\), \(send M \text{ to } y\) is a DL program, also defined below, which does what the send action is meant to do, and similarly for \(acceptCond\) and \(accept\).

\[
\begin{align*}
sendCond(x, M, y) & \equiv \text{class}(x) = EE \lor (\text{head}(\text{todo}(x)) = \text{send } M \text{ to } y) \\
acceptCond(x) & \equiv \text{todo}(x) = () \land \text{size}(\text{in tray}(x)) > 0
\end{align*}
\]

where \(head\) and \(size\) are library functions with the obvious meaning, and \(todo\) and \(in\) are array variables used to represent an object’s outstanding actions and messages respectively. The special class name \(EE\) is introduced so that external entities can be treated as objects having this class.

Note that \(todo\) takes objects to programs, which on the face of it is a category error, because programs are part of the syntax, not individuals in the semantic domain. By adding function symbols corresponding to \(*, :=, \cup\) and \(;\), we can copy the program language into the semantic domain. Whenever you see a program on the right hand side of an equals sign, it is shorthand for a term formed using this vocabulary.

\[^2\text{We make some assumptions about action scheduling and event pool ordering, as suggested by the terms “todo list” and “inray”}\]
Now we define the actions send and accept. When an object $x$ sends a message $M$ to the object $y$, the message is placed in $y$’s intray, and the send action is removed from $x$’s todo list.

$$x . \text{send } M \text{ to } y \equiv$$

$$\text{intray}(y) := \text{append}(\text{intray}(y), M);$$

$$\text{todo}(x) := \text{tail}(\text{todo}(x))$$

where append and tail are library functions.

When an object accepts a message, it makes the state transition specified in its state machine, loads the entry procedure of the new state, and removes the message from its intray.

$$x . \text{accept} \equiv$$

$$\text{state}(x) := \text{nextState}(\text{state}(x), \text{head}(\text{intray}(x)));$$

$$\text{todo}(x) := \text{entryProc}(\text{state}(x));$$

$$\text{intray}(x) := \text{tail}(\text{intray}(x))$$

Where state is an array variable used to record an object’s state. This definition depends on the functions nextState and entryProc, which in turn depend on the state machine diagrams of the model. The definitions of these functions will turn out to be a consequence of the formulae we extract from the state machine diagram. We will consider them to be array variables in order to keep the interpretation general, but any attempt to assign them values that do not agree with the state machine diagram(s) would result in an inconsistency.

This program $\varepsilon$ allows us to say things about model dynamics. Adapting the example in Section 6.1, we can say that nothing bad will happen so long as the model starts within acceptable initial conditions: $\text{Init} \longrightarrow \neg (\varepsilon) \text{Bad}$. Note $\varepsilon$ is a $*$ program, so the program under the $*$ can run 0 times. This means that for $[\varepsilon] \varphi$ to be true in some situation $w$, $\varphi$ has to be true there and in every situation reachable by legal model evolution. Therefore, if we want to say that in our model, $\varphi$ is always true (an invariant), we can assert $[\varepsilon] \varphi$.

This sets the general framework for systems defined by models of our tiny UML subset. Now we are ready to look at the diagrams that define Mellor’s example model.

### Class Diagram

This class diagram (Fig. 5.1 Page 81) does not tell us a lot. It tells us that there are two classes, $A$ and $B$, but this should not be taken as an assertion that every object has one of these classes. There may be other class diagrams that help define this model, or an elaborative model transformation may have introduced new classes. In each of these cases we still wish to consider snapshots of the larger model to be snapshots of this one.

---

3an elaboration only adds details to a model
These two classes don’t even have any attributes. If there was an attribute, we would read it as an assertion that if an object claims to belong to the class \( \text{class}(x) = A \), and has a value for the attribute, then the type of that value matches the type of the attribute in the class diagram. We would not insist that the object have a value for that attribute, because it may not have been set yet \[MB02\ §12.1.2, Page 200\]. Neither would we insist that it only has values for attributes listed in the class, because again, elaborative model transformation may have added attributes to the class, or another diagram might provide part of the class specification. The association however, does tell us something. It says that each object of class \( A \), has exactly one \( \text{ex} \) which is an object of class \( B \). Since this is all the information we can obtain from the class diagram, we will name the formula \( CD \).

\[
CD \equiv \left[ e \right] \left( \forall x \right) \text{class}(x) = A \rightarrow \text{size}(x.\text{ex}) = 1 \land \left( \forall y \right) y \in x.\text{ex} \rightarrow \text{class}(y) = B \)
\]

Notice that we have taken some vocabulary from the class diagram. The class names \( A \) and \( B \) are variables, \( \text{ex} \) and \( \text{class} \) are array variables.

**State Machine Diagram**

State machine diagrams describe behaviour. The diagram itself does not specify what the description applies to. This is what we do when we attach a state machine to a class, we say that objects which belong to this class exhibit the behaviour described in this state machine. As a result, the formulae we obtain from the state machine have a free variable, representing an arbitrary entity. Here, we take the view of Executable UML, assuming that the state machine is attached to a class, not collaborations, or methods \[JRB05\ Page 604\]. We also assume a simplified form of Executable UML’s rules about message delivery and acceptance \[MB02\ §11.2\]. The state machine diagram (Fig. 5.2, Page 81) does not specify which objects it applies to, so the state machine diagram formulae contain a free variable. In Section 6.3, we will use this variable to connect the state machine diagram to the class \( A \).

First, a simple property. For an object to conform to this state machine, it must be in one of the diagrams’ states.

\[
\text{SM}_i(x) \equiv \left[ e \right] \left( \text{state}(x) = s \lor \text{state}(x) = s' \right)
\]

A weaker interpretation is perhaps possible. A state machine diagram, like a class diagram, might be seen as only a partial specification, omitting some detail in order to provide a view for a specific purpose. This seems less plausible than the corresponding example for class diagrams though, so we choose the stronger interpretation.

Now we turn to the transition. The transition in the state machine says that if an object \( x \) is in state \( s \) and it has a \( W \) message at the top of its intray, then after it
does an accept, it will be in state $s'$.

$$SM_t(x) \equiv \epsilon (state(x) = s \land head(intray(x)) = W) \Rightarrow [x.accept] state(x) = s'$$

We will not combine the state and transition formulae yet, but will combine them with the entry procedure for state $s'$ and the class $A$ in Section 6.3.

**Sequence Diagram**

The sequence diagram (Fig. 5.3, Page 81) partly specifies an initial model state, and lists some occurrences in the order that they are meant to happen. It is satisfied by model execution traces that begin in a state that satisfies the diagrams initial conditions, and in which all the occurrences happen legally, in the given order.

Note that other things are allowed to happen in between the occurrences given in the diagram. Read, write, link and unlink actions are not shown in sequence diagrams. An object or external entity not shown in the diagram might send one of the participants a message. Indeed, we might allow participants of the sequence to exchange messages not shown on the diagram. If the diagram is intended as a high-level summary, we might choose to omit some of these details.

The rectangles on the object lifelines are called execution specifications [Obj07c, §14.4, Table 14.1], and they indicate an object activation, which is when an object is doing something. In the Executable UML method, it is conventional to indicate the object state in these rectangles, as we have done in Figure 5.3 on Page 81. All object activations in Executable UML correspond to a state transition. It would be useful in the context of Executable UML to read this as an assertion that the object makes the transition into the indicated state, and perhaps also that it remains in that state until the next transition indicated on the diagram. We will not do that here. We probably should interpret the execution specifications as assertions that the object has a non-empty todo list over the indicated interval. For the sake of simplicity, we will not do that either.

The following formula captures our interpretation of the sequence diagram. It says that $ee$ is an external entity, $a$ has class $A$, $b$ has class $B$, and that it is possible for some stuff to happen, and then for $ee$ to legally send a $W$ to $a$, and then for some more stuff to happen followed by $a$ legally doing an accept (activation) after which some stuff can happen and then $a$ can legally send a $Y$ message to $b$.

$$SEQ \equiv \text{class}(ee) = EE \land \text{class}(a) = A \land \text{class}(b) = B \land \epsilon(\text{sendCond}(ee, W, a) \land \langle ee.\text{send W to a} \rangle) \land \epsilon(\text{acceptCond}(a) \land \langle a.\text{accept} \rangle) \land \epsilon(\text{sendCond}(a, Y, b) \land \langle a.\text{send Y to b} \rangle)$$

As usual, we use variables for all the model specific vocabulary.

It is necessary to insert the action precondition formulae (defined on Page 120) in the appropriate places, because these actions can execute in any situation.
whereas we want to ensure that they only execute when model execution rules permit it.

We do not intend $SEQ$ to be valid, nor even a consequence of $CD$ and $SM$. (The state machine diagram formula $SM$ is defined below, in Section 6.3) We only require that $CD$, $SM$ and $SEQ$ are co-satisfiable. That is, we do not require that in every model snapshot, there is an $a$ of class $A$ and a $b$ of class $B$, and that all this stuff can happen. We only require this to be the case of some model snapshot.

## 6.3 Weaving and Consistency

Having specified the meaning of each of the parts of the model, we now turn to the task of connecting these parts to specify the required system. Because we have interpreted the diagrams using logical formulae, we can join them using logical connectives.

We want to make the send action the entry action of state $s'$. That is, if an object is in state $s'$, then immediately after it does an accept, we want its $todo$ list to contain only this action.

\[ SM_p(x) \equiv [c][x.\text{accept}](\text{state}(x) = s' \implies todo(x) = \text{send } X \text{ to } x.ex) \]

Putting $x.ex$ rather than $self.ex$ saves us the trouble of evaluating $self$.

Recall that in Section 6.2 we defined two formulae from the state machine diagram, $SM_s(x)$ for the states and $SM_t(x)$ for the transitions. Now, we attach the state machine diagram augmented by the entry procedures, to the class $A$.

\[ SM \equiv [c][((\forall x) \text{class}(x) = A \implies SM_s(x) \land SM_t(x) \land SM_p(x))] \]

We are now in a position to ask whether our model is consistent. In other words, is there an execution trace which can satisfy $CD$, $SM$ and $SEQ$?

Semantic tableaux deductive calculi [DGHP99] do a systematic search for an interpretation which satisfies their input formulae. Their purpose is to determine whether or not an argument is valid, that is, that it is impossible for the premises of the argument to all be true in the same situation where the conclusion is false. To test this, the conclusion is negated, and together with the premises, input to the search procedure. If a situation that satisfies these inputs is found, it is a counter-example to the argument, because it makes the premises and the negated conclusion all true, hence it makes the premises true and the conclusion false. It is important that the search procedure is exhaustive, because then if no counter-example is found, we may conclude that the argument is valid. Some logics, such as first order logic and DL are undecidable, so for some inputs, the procedure will not terminate. However, when it does terminate without finding a counter-example, we still know that the argument is valid.

\[ ^4\text{Alwen Tiu pointed out that a decision procedure for DL would be a solution for the halting problem, because for any program } \rho, \text{ the formula } \langle \rho \rangle \top \text{ is valid iff } \rho \text{ halts.} \]
Our goal is different: we want to show that a collection of formulae are consistent. We can therefore simply enter our formulae into the search procedure. If it finds an interpretation which satisfies the inputs, our model is consistent. If it terminates without finding one, it is inconsistent. Since the required execution traces are unlikely to be difficult to find, if the process runs for a long time without terminating, this would be a fair indication that the model has problems.

Tableaux methods for modal logics such as dynamic logic have been intensively studied [DGHP99, vEHN01]. We do not know of a tableau prover for DL with array variables, but the principle author of the Tableau Workbench [AG03] assures us that this system could be easily adapted to the task.

For now, we will conduct a manual search in the style of a tableaux system. The first step is to assume that there is a situation which satisfies $SEQ$, $CD$ and $SM$. We call this situation $w_0$. Recall that these situations are valuations, which take variables to individuals, and array variables to functions. Recall also that programs denote relations between these valuations. Our search will consist in breaking down the formulae until we have a collection of explicitly specified valuations, related as required by DL programs. Or, if the model is not consistent, this inconsistency will be made explicit as two different values for a variable under some required valuation.

We begin by assuming that our three formulae are true at $w_0$.

- $SEQ$ at $w_0$, assumption (6.1)
- $CD$ at $w_0$, assumption (6.2)
- $SM$ at $w_0$, assumption (6.3)

For a conjunction to be true at a world, each of its conjuncts must be true there, so we reduce (6.1) ($SEQ$) to

$$
\text{class}(ee) = EE \text{ at } w_0, \text{ from } 6.1
$$

$$
\text{class}(a) = A \text{ at } w_0, \text{ from } 6.1
$$

$$
\text{class}(b) = B \text{ at } w_0, \text{ from } 6.1
$$

and

$$
\langle \varepsilon \rangle (\text{sendCond}(ee, W, a) \land \langle ee.\text{send } W \text{ to } a \rangle \ldots \top) \text{ at } w_0, \text{ from } 6.1
$$

If we were performing this tableaux search with pencil and paper, we would now mark 6.1 with a tick √ to indicate that it is complete. These four new formulae are satisfied if and only if 6.1 is, so we can forget about it.

Since the class diagram formula $CD$ is $[\varepsilon] \ldots$, and $\varepsilon$ is a * program which may run 0 times, the $\ldots$ part must be true at $w_0$.

$$
(\forall x) \text{class}(x) = A \quad \text{at } w_0 \quad \text{from } 6.2
$$

$$
\text{size}(x.ex) = 1 \land \quad \text{at } w_0 \quad \text{from } 6.2
$$

$$
((\forall y) y \in x.ex \rightarrow \text{class}(y) = B)
$$
The sequence diagram does not actually say that \( b \) is \( a \)'s \( ex \), but this was our intention when creating the scenario, and it makes 6.8 true. We hope that the tableaux system would try this before introducing new instances of \( B \). Even so, it would take several steps to do so, which we omit. Note that there are other ways of making 6.8 true, so we do not mark it as completed at this stage. In particular we could assert the existence of new instances of \( B \) at \( w_0 \). Although it is fairly clear to humans that this would not help, a tableaux system could easily get stuck in an infinite sequence of stupid ideas.

\[
a.ex = \{ b \} \quad \text{at } w_0, \quad \text{from 6.8} \tag{6.9}
\]

Now we turn to 6.7. Again, because \( \varepsilon \) is a \( * \) program, we can satisfy the formula at \( w_0 \) by satisfying the subformula under the \( \langle \varepsilon \rangle \). If this failed, a tableaux system would begin checking other states that can be reached under the \( \varepsilon \) program. We split the two conjuncts.

\[
\text{sendCond}(ee, W, a) \quad \text{at } w_0, \quad \text{from 6.7} \tag{6.10}
\]
\[
\langle ee. \text{send } W \text{ to } a \rangle \ldots \top \quad \text{at } w_0, \quad \text{from 6.7} \tag{6.11}
\]

Now 6.4 says that \( ee \) is an external entity, and the definition of \( \text{sendCond} \), says it can send messages as it pleases, so 6.10 is true at \( w_0 \).

To satisfy formula 6.11 we need a transition from \( w_0 \) to another situation resulting from the send action. We write

\[
w_0 \xrightarrow{\text{send } W \text{ to } a} w_1 \tag{6.12}
\]

to say this. We also assume that \( a \) is in state \( s \) with an empty \( \text{intry} \). This will mean that \( a \) can accept the sent \( W \) message and make the intended transition. Again, a formal tableaux system might have to work to find that this is what is needed.

\[
\text{state}(a) = s \quad \text{at } w_0, \quad \text{assumption} \tag{6.13}
\]
\[
\text{intry}(a) = () \quad \text{at } w_0, \quad \text{assumption} \tag{6.14}
\]

As a result of the send action, \( a \) will have a \( W \) message in its \( \text{intry} \).

\[
\text{intry}(a) = (W) \quad \text{at } w_1, \quad \text{from 6.14, 6.12, df. send} \tag{6.15}
\]

We now reinstate some of 6.11 which we had turned into . . . To make it true at \( w_0 \) we require its subformula to be true at \( w_1 \).

\[
\langle \varepsilon \rangle (\text{acceptCond}(a) \land (a.\text{accept}) \ldots \top) \quad \text{at } w_1, \quad \text{from 6.11, 6.12} \tag{6.16}
\]

The condition \( \text{acceptCond}(a) \) is true at \( w_1 \) because \( a \) has a message in its \( \text{intry} \) but nothing in its \( \text{todo} \) list. To make the other conjunct true, we need an accept step to the next state.

\[
w_1 \xrightarrow{a.\text{accept}} w_2 \tag{6.17}
\]
Now the definition of `accept`, on Page 121, depends on the array variables `nextState` and `entryProc`, so it’s not immediately obvious what the situation will be in `w2`. However

\[ (∀x) \text{state}(x) = s \land \text{head}(	ext{intray}(x)) = W \quad \text{at } w_1, \quad \text{from } SM \ (6.18) \]

and therefore

\[ \text{state}(a) = s' \quad \text{at } w_2, \quad \text{from } [6.13 \ 6.15 \ 6.17 \ 6.18] \ (6.19) \]

It follows from this that, at least at `w1`, `nextState(s, W) = s'`, and since we are not considering any programs which update `nextState`, we can consider this to be the value everywhere. Similarly, at least in effect (since we do not actually use `self`), `entryProc(s') = \text{send} \ X \text{ to } \text{self}.ex` because

\[ (∀x)[x.\text{accept}](\text{state}(x) = s' \quad \text{todo}(x) = \text{send} \ X \text{ to } \text{x}.ex) \quad \text{at } w_1, \quad \text{from } 6.3 \ (6.20) \]

and so

\[ \text{todo}(a) = \text{send} \ X \text{ to } a.ex \quad \text{at } w_2, \quad \text{from } [6.17 \ 6.19 \ 6.20] \ (6.21) \]

Assume that values are equal to their singleton sets, so \(\{b\} = b\).

\[ \text{todo}(a) = \text{send} \ X \text{ to } b \quad \text{at } w_2, \quad \text{from } [6.9 \ 6.21] \ (6.22) \]

Thus we can satisfy **6.16** if we can satisfy

\[ ⟨ε⟩(\text{sendCond}(a, Y, b) \land ⟨a.\text{send} \ Y \text{ to } b⟩) \top \quad \text{at } w_2, \quad \text{from } [6.16 \ 6.17] \ (6.23) \]

and now, it would appear that we are stuck, because it seems nothing that can possibly happen is going to put `send Y to b` into `a`’s todo list, which we need in order to satisfy `sendCond(a, Y, b)`. But recall that `X` and `Y` are variables. Hence each valuation maps them both to individuals in the semantic domain. If `w2` maps `X` and `Y` to the `same` individual, then `sendCond` is satisfied, the `send` can proceed and the sequence successfully completes, showing that it is consistent with the model defined by the class and state machine diagrams.

This is clearly not how the story is supposed to end. We are developing a formal theory of our intuitive understanding of the diagrams, and our intuition says they are inconsistent, because `X` is `X`, `Y` is `Y`, and they can not be the same thing. That is “`X`” is not just a label we stick on that message from outside the system, but rather something essential to its identity. We can capture this idea formally by retrospectively asserting the following naming invariant

\[ \text{NAMES} ≡ [ε](\text{name}(X) = “X” \land \text{name}(Y) = “Y” \land \cdots \land \text{name}(s') = “s'”) \]

which makes \(X = Y\) impossible. It also prevents the classes `A` and `B` from being the same, and also weird possibilities like `A = s`, identifying a class and a state.

127
(Although Mellor [Mel03] has made suggestions that are almost this strange when discussing how domain models might be woven together.)

So now there really is no way to find a situation with send $Y$ to $b$ on $a$’s todo list, so the sequence diagram is inconsistent with the model. This could be proven by induction on the definition of $\varepsilon$.

6.4 Interactions as Dynamic Formulae

This chapter has presented a formalisation of a UML fragment including the simplest elements of class diagrams, state machines, actions and sequence diagrams. The formalisation has its weaknesses. It directly treats diagrams rather than UML’s abstract syntax. The semantics are derived from our reading of [MB02] rather than the official definition. Dynamic logic lacks types and specialisation, and we have not attempted to encode them. The DL variant used in [WS96, WB97] has ordered sorts for this purpose. We have no concurrency, though concurrent dynamic logics have been studied [Pel87]. Readability must also count as a serious weakness of dynamic logic formulae.

Despite all this, the basic principles underlying this work suggest an alternative to UML’s treatment of Interactions. In this chapter, a UML system state is a valuation of variables, including some “array variables” which are effectively updateable functions. One action and one form of causation were defined as dynamic logic programs, that is, modal operators. We used these to define legal system evolution, so that we could make assertions of the form, “after [any|some] legal evolution of the system, such and such is the case”. These definitions of action and causation were also used as OccurrenceSpecifications in the Interaction. Event occurrences here are not persistent entities added to the system state to record its history. Rather they are the transitions between system states. Instead of looking at the recorded history after the fact, the dynamic logic formula version of the Interaction looks at the possible evolutions of the system.

How could we integrate this version of Interaction semantics into the graph transformation based UML semantics we have outlined in previous chapters?

A graph transformation system would be generated by rules, one for each concrete UML action and kind of causation such as accepting an operation call. Each of these rules would have a name, which would be used to label the transitions in which the rule was applied. Some transitions would be labelled with several rule names, since several things can happen concurrently. Each rule application is determined by a match, a graph homomorphism that chooses nodes in the system state to play the roles defined by the rule. Key participants could also be included in the transition label, so for example, a transition may be labelled $a$ send $y$ to $b$, where $y$ is a Signal instance and $a$ and $b$ are objects.

These transition labels could be the atomic programs of a UML specific dynamic logic. So for example, the Interaction of our example model could represent the formula
We quantify the run-time participants, but not the Signal $Y$ which is part of the model. Sequential composition is not always appropriate, since the ordering of event occurrence specifications in an Interaction can be properly partial, as we saw in Section 5.4. One solution would be to include a set composition to the language. This would apply to a set of programs, and would be satisfied by a path that includes each of them in any order. “Includes” rather than “consists of” because as we pointed out in the body of this chapter, sequence diagrams should not be expected to spell out everything that happens. Another option would be to define a partial order composition operator to directly capture Interaction semantics.

These two possibilities would be executed by writing an interpretation function which would examine the relevant model elements and yield a formula of some graphical dynamic logic. That logic could also be used as a constraint language. It may be possible to instead define the value of an Interaction in a given system state directly, as we did in Chapter 4 with Property. A similar approach could be applied to Multiplicities, but semantics for OCL would remain a problem.

Event occurrences are not only “truth-makers” for Interactions. Recall that in Section 5.3 we found that objects maintain a “pool” of event occurrences, which they may respond to at some later time. This role is not so easy to reconcile with our proposal that event occurrences should be system state transition labels. Perhaps these persistent entities in the system state should instead be considered as notifications of event occurrences. Thus, when they are “consumed” they can disappear. It also makes better sense of occurrences such as TimeEvent instances. Although the time changes constantly, there are only certain circumstances when it is required to notify some object about it.

There are clearly many details to be worked out, but it seems that a dynamic logic approach as we have outlined might be a way to make much better sense of Interactions than the current definitions recorded history approach.
Chapter 7

The UML Formalisation

Literature

UML models are not clearly understood by their users. There are two reasons for this. Firstly the semantic domain of the language is only partially and somewhat inconsistently defined. Secondly, the definition is very complicated and difficult to read. What is needed is a complete, precise and easy to understand definition that agrees with the current one.

We have been carefully studying the language trying to clarify what models mean. In Chapter 3 we noticed that models and system states consist of things, or “objects”, with named values attached to them. We also noticed that the core of the metamodel is actually a mathematical definition for exactly this kind of structure: an edge-labelled directed multi-graph. Apart from the previous chapter which used dynamic logic, we have held the working hypothesis that UML systems are graph transformation systems.

This hypothesis has been able to explain everything we have encountered so far, but a hypothesis which works is not necessarily the best hypothesis. In this Chapter, we survey the wide variety of work which has attempted to give precise meaning to UML, or to parts of it. It is important to note that a lot of this work does not explicitly set out to solve the UML definition problem as we have conceived it. Often the contributions seek to apply specific tools to specific verification problems and as a byproduct, give a formalisation of part of the language.

The first section presents the well known relational model and entity relationship model from database theory, not as possible UML formalisation techniques, but as examples of conceptual clarity that UML should aspire to.

Each of the following sections are centred around a particular formalism. Each begins with a short introduction to that formalism, and outlines the ways in which it has been used. We then examine in some detail two or three of the most significant papers that apply that technique. A concise survey of the remainder of the relevant literature is then followed by an assessment of this technique according to the criteria we developed in Chapter 2. These criteria are briefly recalled in Table

130
0. **faithful** to existing UML definition

1. **understandable** and implementable (supports agreement and tools)

2. **defines** concrete syntax → abstract syntax → semantic domain

3. **clarifies** ideas: model consistency, model refinement, transformation soundness

4. **reflective** - objects can access their classes

5. **flexible** to support profiles and variations

Table 7.1: Summary of criteria for an improved definition of UML

<table>
<thead>
<tr>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>faithful</strong></td>
</tr>
<tr>
<td><strong>understandable</strong></td>
</tr>
<tr>
<td><strong>defines</strong></td>
</tr>
<tr>
<td><strong>clarifies</strong></td>
</tr>
<tr>
<td><strong>reflective</strong></td>
</tr>
<tr>
<td><strong>flexible</strong></td>
</tr>
</tbody>
</table>

Having applied these criteria to all the main formal approaches, the next chapter recommends with confidence, the way UML ought to be defined.

**7.1 Relational and Entity-Relationship Models**

The relational model for databases has been a huge success. It was introduced by Codd in 1970 [Cod70] and is very widely used and taught using texts such as [EN07]. Like UML, the relational model provides concepts for representing information. Unlike UML, those concepts are simple, clear and few. The entity-relationship model [Che76] launched the field of conceptual modelling from which UML grew, and is possibly still more widely used than UML [DGR+05]. In this section, we review the relational and entity-relationship models, both as examples for UML to aspire to, and for their possible application to UML semantics.

The relational model provides a uniform way of presenting stored data to its users, independently of the way it is stored and accessed. It is a convenient halfway point between the conceptual frameworks of the computer system and the real-life situation it is intended to represent. The implementation of a relational database management system (dbms) reconstructs the relational ideas using the computer and operating system level ideas. The entity-relationship model was introduced so that real-life situations could be represented more directly, using an intuitive visual language. Those higher level representations can then be translated into relations, or another lower-level representation. These three conceptual levels and their connections are depicted in Figure 7.1.

In the relational data model, information is represented as a collection of mathematical *relations*. The database version of relations is slightly more elaborate than the usual mathematical one, to accommodate the names needed for practical database work.

A *domain* is a set of values and a *relation schema* is a named set $R = (\text{dom}(A))_{A \in R}$.
of domains indexed by \textit{attribute} names. This is more often written as

\[ R(A_1 : \text{dom}(A_1), \ldots, A_n : \text{dom}(A_n)), \]

and the attributes considered as ordered from 1 to \(n\). The \textit{relation states}, or simply \textit{relations} of this relation schema are the subsets of \(\prod_{A \in R} \text{dom}(A)\) or

\[ \text{dom}(A_1) \times \ldots \times \text{dom}(A_n). \]

These two sets are usually considered equal in database theory. A \textit{database schema} is a collection of relation schemas, and a \textit{state} of a given database schema is a collection of named relation states, one for each of the relation schemas.

The tuples in these relations often represent some real world entity, with each attribute value in the tuple representing some property of that thing. The kind of things represented by a relation may be uniquely identified by some property or properties, such as a student number, or the combination of post code\footnote{In the USA, this is called a “zip code”}, street name and number. In this case we want to avoid having distinct tuples with those attribute values in common, because this would be contradictory information about the thing represented.

This difficulty is catered for by imposing a \textit{constraint}, a condition restricting the set of database states that are considered \textit{valid} instances of the database schema. A \textit{key} constraint specifies a subset \(K\) of the attributes of a relation schema \(R\), and asserts that if for a pair of tuples \(r, t\) in a state \(R\) of \(R\) we have \(\pi_k(r) = \pi_k(t)\) for each attribute \(k \in K\), then \(t = r\). That is, we can not have distinct tuples that agree on the key. Another way of looking at this is that the relation becomes a partial function from the key to the remainder of the attributes. It is usual to nominate a single \textit{primary key} for each relation schema.

To extract the required information from a database, it is often necessary to \textit{join} relations. To find a student’s timetable for example, the tuple representing that student must be connected to tuples representing the courses she is enrolled in, and
these to the lecture time-slots associated with those courses. To achieve this, relations often have attributes which refer to the primary key of another relation (or the same relation). For example an enrolment relation may have a studentID attribute referring to the primary key of a student relation, and a courseCode attribute, referring to the primary key of a course relation.

A join $R \bowtie Q$ is defined to be the subset of the cartesian product $R \times Q$ that satisfies the condition $c$. The product of two relations, in database theory, is the “flattened” form, so that for example $\{(w,x)\} \times \{(y,z)\} = \{(w,x,y,z)\}$ rather than $\{((w,x),(y,z))\}$. The condition $c$ is a statement which can use the attributes of $R$ and $Q$ as variables. When the same attribute name occurs in both relations, the relation name is used as a qualifier. So for example, part of the solution to our timetable query might be $\text{student} \bowtie_{\text{student.studentID} = \text{enrolment.studentID}} \text{enrolment}$. Now since $\text{studentID}$ is the primary key of the $\text{student}$ relation, there is no more than one student tuple for each enrolment tuple. If there is an enrolment tuple with a $\text{studentID}$ value that does not occur in the student relation, this means that a non-existent student is enrolled in some course.

Such a database state is invalid, and referential integrity constraints are employed to exclude them. We impose referential integrity constraints by declaring a list of attributes $X$ in one relation schema $R$ to be a foreign key, which refers to a similarly typed list of attributes $Y$ in another (or the same) relation schema $Q$. For a state to be valid, for each $r \in R$ there must be some $q \in Q$ such that $\pi_X(r) = \pi_Y(q)$.

Each condition $c$ defines a unary relational operation $\sigma_c$. This is called select, because it selects the tuples which satisfy the condition. Since each tuple $r \in R$ is a valuation of the attribute names of the relation schema $R$, we write $r \models c$ when that valuation makes $c$ true. The selection operation then is defined by $\sigma_c(R) = \{ r \in R \mid r \models c \}$. Thus the join $R \bowtie Q = \sigma_c(R \times Q)$. We may also rename a relation $R(A_1, \ldots, A_n)$ and its attributes, using the notation $\rho_{Q(B_1, \ldots, B_n)}(R)$ where $Q$ is the new relation name and the $B_i$ are the new attribute names. These operations $\sigma$ and $\rho$, together with relational product $\times$ and its projections $\pi$, and the usual set operations $\cup$, $\cap$ and $-$ (set difference), are collectively known as relational algebra. This is not to be confused with Tarski’s relation algebra [Tar41].

The standard relational database query language SQL [EN07, Chapter 8] is mostly a sugared version of relational algebra. This not only gives the language a precise semantics, it is also of great use in evaluation of its queries. Database systems evaluate SQL queries by converting them into relational algebra terms. These terms are rewritten using laws of relational algebra to produce a form that will generally execute more quickly. A range of equivalent terms will sometimes be produced, and their processing cost estimated using statistical information about the database. The form with the least estimated cost is then chosen for execution.

The mathematical definition of the relational model has also enabled the development of a theory of functional dependencies and normal forms, which provides a measure of schema quality widely used in database design processes.

The relational model provides a simple uniform way of organising and manip-
uating data, insulating the user from the variety and complexity of storage and implementation. However, there remains a considerable conceptual gap between mathematical relations and the way we see those parts of the world that we want to represent in databases. Chen [Che76] distinguishes between four levels of view, the highest being “information concerning entities and relationships which exist in our minds.” The relational model, he says, is concerned with the middle two levels “organisation of information in which entities and relationships are represented by data” and “the data structures which are not involved in search schemes, indexing schemes, etc”. The lowest of the four levels is the “access-path-dependent data structure”.

The entity-relationship model represents entities and relationships directly as data value sets and relations. Note that Chen carefully describes his data model as a mathematical structure. The much more widely known visual notation of entity relationship modelling is given a precise meaning in terms of this mathematical data model. Never the less, it directly addresses the highest level “view” of information, the entities that we recognise in the world and relationships that exist between them. Chen also describes possible translations from his data model into the relational model.

The relational and entity relationship models can be seen as ancestors of UML, and UML ought to aspire to the conceptual clarity and widespread use that these models enjoy. One of the earliest popular object oriented modelling methods, the Shlaer-Mellor method [SM88] [SM92] employed a modest extension of these models, adding state machines to describe object lifecycles. UML has its own somewhat defined conceptual model. To what extent can this model be reduced to the relational or entity-relationship models? That is, could UML system states be defined as database states?

One could begin by translating class diagrams as the corresponding concepts of the entity-relationship model are translated, treating classes as entity sets, UML attributes as entity-relationship attributes and associations as relationships. This translation yields a relation for each class and a relation for each association with * at both ends. Where one end has a multiplicity with a finite upper bound, the association can be translated as attributes in the relation representing the class at the far end of that association. The “extended entity-relationship model” incorporates a notion of generalisation with a translation into the relational model [EN07, Chapter 7]. This is the kind of translation that is employed when UML is used to design relational databases.

UML’s object oriented model is not the same as the entity-relationship model. The identity of an object does not depend on its attribute values. To capture this in the relational model, an object identifier attribute would need to be added to each relation representing a class. As we discussed in Chapter 4 the semantics of UML associations remains undecided. Depending on the outcome of that debate, it may be necessary to include a separate identifier attribute to relations translated from associations too. The primary key of a relation obtained from a binary relationship is usually the two foreign keys referring to the entity relations at its ends. This
Figure 7.2: Normalising a list valued attribute

does not allow the possibility of a pair of entity/objects having more than one link between them from the same relationship/association.

Whilst object and link identity can be handled by simply adding identifier attributes, there is a much deeper conceptual mismatch between the UML and relational notions of attribute. Codd [Cod70] gave a definition of first normal form (1NF) which says that attribute values are “simple”, that is, neither sets nor tuples. This condition is now considered, in database theory, to be part of the definition of relation itself [EN07 §10.3.4]. On the other hand, UML’s attributes are Properties [Obj07b §7.3.44], whose values can be sets, ordered sets, bags or sequences, depending on the settings of the meta-attributes isOrdered and isUnique. Representing these kinds of values in the relational model would require normalisation.

Consider the most general case, of a sequence valued attribute. An example showing normalisation of such an attribute is shown in Figure 7.2. Underlining is used to indicate each relation’s primary key.

The normalisation is similar to the standard one for set valued attributes, but adds an attribute to record the position of each value in the list. The set normalisation would make id and P the primary key, thus preventing the same value from occurring twice for the same object. The same value can occur twice in a list, but the same position can not, thus we have included pos in the primary key. This still does not ensure that we have a list because there may be “holes” in our sequence. That is, we must enforce a rather awkward integrity constraint, that the values of the new attribute pos are an initial segment of the positive integers, \{1, 2, \ldots, n\} for some n.

When designing a database, one would of course translate each attribute or association according to its special properties, minimising the number of relations in the resulting schema and the number of join operations required when querying it. It is desirable however, that a formal semantics be simple and uniform, to simplify our reasoning about it. Complexity in the statics would also lead to additional complexity in the dynamics. For example, if different kinds of Properties had different semantics, the formal definition of the WriteStructuralFeatureAction [Obj07b §11.3.54] would involve multiple cases.

A relational semantics for a given UML model would therefore contain a lot of relations like the one at the far right of Figure 7.2. It is not hard to see that this relation is equivalent to a linearly ordered set of three P labelled edges leaving the node 1, which is how we formalised Properties in Chapter 4.
We have outlined the relational and entity-relationship models employed in the database field. These models are firmly based on simple mathematical ideas. This conceptual clarity facilitates understanding and hence agreement, as sought by our Criterion 1. The mathematical foundation also enables useful theory and tool support. For example, relational algebra enables query optimisation, and functional dependencies and normal forms allow objective assessment of schema quality and automated support for the design process. UML should aspire to this level of clarity, but we found that it is not practical to directly employ the relational model to do this. In particular, UML’s object identity and collections do not fit well in the relational model.

### 7.2 Graph Transformation

Since Chapter 5, it has been our hypothesis that the semantic domain of UML is graphs, and in Section 5.6 we began to consider graph transformation for UML’s dynamic semantics. A great many other workers have come to this conclusion before us, and in this section we introduce graph transformation more carefully, then study and assess the contributions that apply this theory to the formalisation of UML.

At school we learn rules that can be used to simplify algebraic terms, for example, \( x \times (y + z) = (x \times y) + (x \times z) \). The same technique can be applied for computation in all kinds of abstract algebras using term rewriting systems \[BN98\]. The algebraic terms can be used to model system states, with the rewrites representing state transitions.

Now terms are trees, with each \( n \)-ary function symbol occurrence as a node with \( n \) outgoing edges. Each of these nodes also has one incoming edge, except for the outermost function symbol occurrence, which is the root of the tree. A tree is a directed graph, with a unique path from the root to each node. Graph transformation extends this rule based updating of these special kinds of graph, to graphs in general.

A term rewriting rule is just an equation, such as \( x \times (y + z) = (x \times y) + (x \times z) \), but considered as directed from left to right. To apply the rule, we find an instance of the left hand side somewhere in a term, and replace it with the right hand side, instantiated in the same way. These ideas can be defined with mathematical precision \[BN98\], which we shall outline in the following example.

The term \( 15 + (3 \times ((2 \times w) + 4)) \) may be rewritten to \( 15 + ((3 \times (2 \times w)) + (3 \times 4)) \) using our example rule, because the subterm \( 3 \times ((2 \times w) + 4) \) is an instance of \( x \times (y + z) \). In particular, the substitution \( \sigma = \{ x \mapsto 3, y \mapsto (2 \times w), z \mapsto 4 \} \) turns the left hand side of the rule into the subterm, and turns the right hand side of the rule into the replacement, \( (3 \times (2 \times w)) + (3 \times 4) \), of that subterm. In formal definitions such as those in \[BN98\], each node of the original tree, and hence each subterm of the original term, is identified by its position in the tree.

This is rather simpler than the general graph case. The left hand side of the term...
rewriting rule has a root node, and this must be matched with a single position in the input tree. The subtree rooted at that position is removed, and replaced with the substituted version of the right hand side of the rule. The two sides of a graph transformation rule are both general graphs. There is no distinguished node which guides how it can match the input. Each node of the left hand side must be mapped to some node in the input, and the edges mapped in agreement with the mappings of their source and target. Such a mapping of graphs is called a graph homomorphism, which we defined in Definition 5 (on Page 45). The parts that need to be removed from the input graph, are those which match parts of the left hand side of the rule that do not occur on the right hand side. Similarly, the parts that must be added to the input graph are those nodes and edges from the right hand side of the rule that do not occur on the left.

There are several alternative ways of defining these ideas formally, but we focus on the one that seems most popular in the UML related literature, and which is treated thoroughly in the recent monograph [EEPT06]: the double pushout method. The outline which follows is mostly drawn from this book. Pushout is an idea from category theory which is used in this method to define what it means to “glue” two graphs together. This is obviously helpful for adding the nodes and edges from the right hand side of the rule, but it is also used to remove parts of the graph by “ungluing” them.

Let’s say we have two graphs $B$ and $C$ which we wish to “glue together” forming a new graph $D$. We must also specify which nodes and edges in $B$ are to be glued to which nodes and edges in $C$. The resulting graph $D$ will “contain” copies of both $B$ and $C$, but some $D$ nodes and edges will be in both copies. These nodes and edges at the intersection of the input graphs are a subgraph of the resulting graph, showing where the gluing has occurred.

A pushout is a special “bottom corner” for a given “top corner” of functions. This is shown at the left of Figure 7.3 with the given top corner shown as solid arrows, and the pushout bottom corner shown with dashed lines. In fact, the arrows in category theory need not be functions, and the objects at the ends of the arrows need not be sets or set based structures like graphs. We will ignore this level of generality, and just consider the case where the objects are graphs and the arrows are graph homomorphisms. There are a couple of conditions for $f', g', D$ to be a pushout. First, the square must commute, which means that the top-left path must give the same results as the right-bottom path, that is $g' \circ f' = f' \circ g$. Secondly, the pushout must be universal, or limiting, in the sense that, for any other corner that makes the square commute, there must be a unique arrow from the pushout to this other corner. This idea is pictured in the right hand side diagram in Figure 7.3. The arrows $h$ and $k$ and the graph $X$ are an alternative bottom corner forming a commuting square. By saying that our pushout $f', g', D$ is universal, we mean that for any alternative corner, such as the one shown, there is a unique arrow, in this case $x$, which converts the pushout into that corner. That is to say, $x \circ f' = k$ and $x \circ g' = h$.

Now, here is what we do if we want to glue two graphs together. We call one
of the graphs $B$ and the other $C$. We make another graph $A$ which shows the glued intersection of the resulting graph. For each node in $A$, we show which node in $B$ and which node in $C$ will be at that place, by setting the value of the functions $f$ and $g$ respectively, and similarly for the edges. We then construct a pushout over this corner, and the resulting graph $D$ will be the glued together result that we require.

This is proven, though disguised in rather technical language, in [EEPT05, Fact 2.17]. In particular, it shows four things, each of which is part of what we intuitively expect of a glued together graph. We will take a brief look at each of these four points, explaining why the stated property is desirable. Each of the quotes are from the Fact just cited.

“If $f$ is injective (or surjective), then $f'$ is also injective (or surjective).” We are using $f$ and $g$ to show which pairs of nodes and edges to glue together, with each pair having one element from $B$ and one element from $C$. If $f$ was not injective, then two elements of $A$ are mapped to the same element of $B$, and thus both would be glued to some element of $C$ and hence to each other. This amounts to gluing the graph to itself rather than just to another graph, so $f$ and $f'$ are indeed injective. That $f'$ is injective means that $D$ contains a complete copy of $C$. Similarly, because $g$ and hence $g'$ are injective, $D$ contains a complete copy of $B$.

“The pair $(f', g')$ is jointly surjective, i.e. for each $x \in D$ there is a preimage $b \in B$ with $g'(b) = x$ or $c \in C$ with $f'(c) = x.$” This means that there is no “junk” in the resulting graph $D$. Everything there is copied from $B$ or $C$ or both.

“If $f$ is injective and $x \in D$ has preimages $b \in B$ and $c \in C$ with $g'(b) = f'(c) = x$, then there is a unique preimage $a \in A$ with $f(a) = b$ and $g(a) = c.$” Recall that we ask for a pair of things to be glued together by mapping some element of $A$ to the two elements of the pair. So this is telling us that the only pairs of things which have been glued together are the pairs of things we said to glue together.

“If $f$ and hence also $f'$ is injective, then $D$ is isomorphic to $D' = C \uplus B \setminus f(A)$.” That is, the resulting graph is just like the disjoint union of the two input graphs, except there is only one copy of $A$, which is because the two copies have been
Also note that any two pushouts for the same top corner are isomorphic [EEPT06, Fact 2.20]. If this were not the case, then using pushouts to define rule application would be ambiguous.

Let us now see how pushouts are used to define graph transformation rules and their application. Intuitively, a graph transformation rule is a pair of graphs, before and after. In order to define rule application as a pair of pushouts or gluings, rules are seen as a pair of graphs, along with the inclusion maps from their intersection. That is, if \( R \) is the before graph (right hand side) and \( L \) is the after graph (left hand side), then the rule is \( L \leftarrow K \rightarrow R \), where \( K \) is the graph which contains only nodes and edges that appear in both \( L \) and \( R \). This enables rule application to be defined by the double pushout diagram shown in Figure 7.4.

The match \( m \), at the left hand side of Figure 7.4, is a graph homomorphism from the left hand side of the rule \( L \) into the graph to be transformed \( G \). The result of this rule application is obtained in two steps, each step being the completion of a pushout.

The first step is to find the graph \( D \) and morphisms \( k \) and \( f \), such that \( f, m, G \) is a pushout. That is, we want to find a minimal graph \( D \) from which we can obtain the original graph \( G \) by gluing in the left hand side of the rule \( L \). Because of the pushout properties we discussed above, it will contain the intersection of the left and right hand sides of the rule, that is, the rule context which is neither added nor deleted, but it will not contain the elements of \( L \) that are not in \( K \). Since \( m \) may not be injective, \( k \) might map distinct nodes or edges in \( L \) to the same node or edge in \( D \). Thus, \( D \) is \( G \) without the parts that the rule deletes.

The second step is to find a pushout under \( D \leftarrow K \rightarrow R \). That is, glue \( R \) into \( D \) at the places shown by these arrows. The result is \( H \), which is \( G \) transformed by the rule at the match \( m \).

Transformation systems have been studied for many variants of graph, directed and undirected, with and without parallel edges, and with and without node and edge labels and “attributes”. In each case, the graphs and relevant form of morphism between them form a category in which the gluing pushouts are constructed. To capture the notion of model instantiation, another kind of graph category is employed. A graph \( TG \) is nominated as the type graph, which plays the role of a class diagram. Then the object diagrams or system states which instantiate the...
class model are represented by pairs consisting of a graph $G$ and a homomorphism $g : G \rightarrow TG$. These are the objects of the category of graphs typed over $TG$. An arrow $(F, f) \rightarrow (G, g)$ in this category is a graph homomorphism $h : F \rightarrow G$ which makes the triangle of Figure 7.5 commute, i.e., $f = g \circ h$. (This is an example of a slice category, which is in turn a simple form of comma category [BW90, Pages 3 and 13].) This just means that if $h$ maps one object to another, they must have the same class.

The obvious starting point in the graph transformation literature is Baresi and Heckel’s “Tutorial Introduction to Graph Transformation: A Software Engineering Perspective” [BH02]. The ideas of graph transformation are introduced using only a little basic mathematics, and the many variations of graph and graph transformation are outlined. This is followed by a survey of the literature on applications to software engineering, mostly related to the syntax and semantics of visual languages. Type graphs are explained as a formalised version of metamodels. This is an advance over other formalisation techniques, which abandon metamodeling, replacing it with traditional grammar. Two ways are distinguished of using graph transformation to define semantics. The first is to write rules, once and for all, defining a “language interpreter”. The paper of Engels, Hausmann, Heckel and Sauer [EHHS00] is cited as an example of this. The second way is to define a translation, which takes a model, and yields rules defining a transition system for that model. The work of the Martin Gogolla’s “Bremen school” [ZHG05, KG01] follows this plan. We discuss both approaches below.

Functional and architectural models are distinguished, and graph theoretic techniques are shown for integrating them. Concrete syntax, abstract syntax and the semantic domain can all be defined using graphs, and important relationships between these can be defined using graph transformations. For example, parsing of concrete diagrams into abstract syntax, layout of abstract syntax to yield concrete diagrams, editing of abstract syntax using a graph grammar, and visualisation of model execution. There are sections on the available software tools, and on theoretical work such as concurrency and term graph rewriting for efficient evaluation of functional languages. Many of the main ideas are demonstrated by examples, and the bibliography is very extensive. This tutorial is an excellent introduction to graph transformation for visual language semantics.

---

2 so called by Hausmann [Hau05] of Paderborn
The recent monograph [EEPT06] consolidates and presents research on the double pushout approach to graph transformation systems. It mentions UML, and gives many examples using graph transformation systems to model computer systems, but does not directly address the problem of defining UML’s semantics. The first of its 4 parts outlines graph transformation systems in a relatively straightforward algebraic way, using category theory only as needed to define rule application by pushouts. The second introduces adhesive high-level replacement categories, which admit the common kinds of graph transformation systems as special cases. The third treats the author’s notion of “attributed graph” (there are competing notions). They claim this rather complicated graph variant is suitable “to specify and implement visual modeling techniques such as UML” (Page 171), but give no justification. This kind of graph transformation system is implemented in their tool AGG, which is the topic of the fourth part. A case study is presented of a model transformation between UML statecharts and petri nets.

The other standard reference on graph transformation is the 3 volume handbook [REE+99]. This is broader in scope, covering all approaches to defining graph transformation steps, but a little less up to date. Manfred Nagl has been a prominent contributor to the graph transformation literature from the mid 1970’s [Nag76] to the present day. Another prominent worker in this field is Andy Schürr. Some of Schürr’s work addresses aspects of UML, or is applicable to it [ES95], but we have not seen anything from Schürr or Nagl that suggests an approach to defining complete semantics for the language.

That completes our introduction to graph transformation and the more general literature on it. There are two distinct well developed approaches to applying graph transformation theory to the formalisation of UML, each supported by several publications. The next subsection studies the most promising of these approaches. The other major body of work is covered in the following subsection, along with a few other graph related contributions.

**Dynamic Metamodelling**

Dynamic metamodelling is an approach to the definition of UML developed at the University of Paderborn primarily by Jan Hausmann, Gregor Engels and Reiko Heckel [EHHS00, EHHS02, Hau05, ESW07]. It draws on theoretical work of Andrea Corradini and Ugo Montanari of the University of Pisa in collaboration with Heckel [CHM00]. The UML metamodel is written in a subset of UML which mainly consists of the simpler parts of class diagrams. Dynamic metamodelling extends the metamodelling language to include operations and collaboration diagrams. The operations are used to represent the things that can happen in a UML defined system, such as an object sending a message, triggering a state machine transition and so on. The operations are defined by graph transformation rules, and system state transitions are labelled with these operations when they are generated by the corresponding rule. The rules are formulated using UML collaboration diagrams. The parts of the system state that are to be removed are marked in the
diagram with \{\textit{destroy}\} and the parts to be added marked \{\textit{new}\}. Also added to the metamodel are classes describing the auxiliary elements of the run-time semantic domain, such as activity executions and tokens. The metamodel then acts as a type graph for a graph transformation system. Thus a formal semantics for the language can be specified by a metamodel using this slightly expanded sublanguage of UML.

This approach was first presented in \[\text{EHHS00}\]. The semantics are applied to the model consistency problem in a surprising way in \[\text{EHHS02}\]. Rather than take a model and answer the question, “is it consistent?”, this approach takes two separate models along with a specification of “consistency constraints” and tests whether the two models are consistent with one another, in the sense specified. We accept that different phases of development have different consistency requirements, determined by whether the model is a partial or complete description, but dividing a model into two so that an inconsistency can be found seems almost as hard as simply finding the inconsistency. Hausmann’s PhD thesis \[\text{Hau05}\] works through the DMM proposal in detail, applying it to a fragment of UML activity diagrams which he then analyses using the graph transformation system model checker Groove \[\text{Ren03}\]. (Model checking is discussed in Section 7.3 on page 148.) A similar study of activity diagram formalisation using DMM is presented in \[\text{ESW07}\]. Here the aim is to automatically test the activities against criteria taken from the literature on workflows. These criteria are coded into computational tree logic (CTL) so that Groove can check whether they are satisfied by the transition system specified by the example activity diagram.

Hausmann’s thesis \[\text{Hau05}\] is very similar to the present work in its aims and its conclusion. We therefore take a moment to outline that work and compare it with our own. The first chapter introduces software modelling and the problem of informally defined semantics. In the next chapter, he distinguishes between concrete syntax, abstract syntax, and semantics, and briefly examines the special features of UML and similar languages. Similarly to our Chapter 2 (published as \[\text{OK06}\]), Hausmann develops criteria for a definition technique, then applies these criteria in a survey of UML formalisations. We both require that a formalisation be precise and understandable, support analysis by tools, define the complete language and be flexible or “universal”. The survey is admirable for its breadth, but writes off some major contributions such as \[\text{RCA01}\] with a one line table row, because they don’t explicitly offer a formalisation of the whole language. We prefer to ask “\textit{could} this technique define the whole language?” The remaining contributions, he classifies into compilational, operational and hybrid semantics.

In fact, he refers to “denotational or compilational” semantics and includes \[\text{CEK01}\] under this heading, even though those authors say that “the semantics described in this paper is imperative and operational”. In a denotational semantics “the meaning of a well-formed program is a mathematical function from input data to output data” \[\text{Sch96}\]. Such a semantics for UML would simply not be a good idea. Denotational semantics deliberately ignore program behavior, but for UML, behavior is at least as important as functional computation. What the contributions
under this heading have in common is that they translate UML into another language which has well defined semantics, which is why he calls it “compilational”. Like many authors, Hausmann mistakenly calls this target language the “semantic domain”. If the semantics of language \( L \) is defined by translation into first order logic, then the semantic domain of \( L \) is not first order logic but the semantic domain of first order logic, namely the class of Tarski-structures.

Included under the “denotational” heading is “denotational metamodelling”, which are approaches that give a model of the semantic domain, and a mapping of the syntax into it \([\text{KGR99}]\). We noted in Section 5.3 that the UML definition now contains fragments of a model of its semantic domain \([\text{Obj07c}, \S 13.1]\), although it is only there to explain rather than to define. Many authors refer to such a model as a “metamodel”. This is fair enough, in that it is a model of the things that we will use to model the world. The UML “metamodel” is really a prescriptive model of the language we use to describe those models. The “meta” is perhaps justified by our argument in Chapter 1 that instances of the UML abstract syntax actually are models as well as descriptions. In any case, we should take care to distinguish between models of the language and models of the semantic domain.

The DMM authors \([\text{Hau05, ESW07}]\) take dynamic metamodelling to be a hybrid of this “denotational metamodelling” approach, which covers the static semantics, and the operational technique of graph transformation to handle the dynamics.

The semantic mapping aspect of existing denotational metamodelling work seems inadequate to Hausmann, so he has developed the notion of “metarelation” to handle it. This is the topic of his third chapter. Neither the problem nor the solution seem very clear to us. The following chapter introduces graph transformation systems, and then Chapter 5 describes the technique of dynamic metamodelling in detail. The remaining chapters and appendices give a case study and practical guidelines for the application and automation of DMM.

The aim of the thesis is a method for defining UML-like languages, and the contribution is a detailed proposal, which in our opinion is the best currently available. In contrast to our own work, Hausmann moves quickly over the nature of the problem to be solved and the details of UML itself, and gets on with exposing the new ideas. Our contribution is more analytical and critical. We have studied the nature of modelling, carefully examined the existing UML definition and have now embarked on a thorough survey of the formalisation literature. Our conclusions in Chapter 8 will not be so detailed as Hausmann’s but they will be better supported, and thus a better foundation for further work.

All of the dynamic metamodelling work is based on the theoretical foundation of graphical operational semantics (GOS), which is presented in the brief workshop paper of Corradini, Heckel and Montanari \([\text{CHM00}]\). It extends traditional graph transformation systems by introducing rules which have premises and a conclusion, in the style of Plotkin’s structural operational semantics (SOS) \([\text{Plo81, AFV01}]\).

As we saw earlier in this section, a term rewriting rule is just a directed equation, a before term \( L \) and an after term \( R \), perhaps with some shared variables. We could write such a rule as \( L \rightarrow R \). Term rewriting systems allow rules to ap-
ply to subterms, which in their usual algebraic applications makes sense, because if \( L = R \) then \( f(L) = f(R) \). SOS rules must match the whole term, which makes sense for sequential programming languages, because you want the first instruction to be evaluated first. This restricted application to subterms is controlled by rules with premises. So in our sequential programming language example, we might have a rule

\[
\begin{align*}
    x & \rightarrow x' \\
    x; y & \rightarrow x'; y
\end{align*}
\]

which only allows evaluation of the first subterm. Similarly, SOS allows specification of the evaluation order of terms such as \((w + x) + (y + z)\). A term rewriting system would have distinct paths from this expression to its value, one evaluating left subterm first, the other the right. Another advantage of SOS specification is that it allows the definition of transition systems with larger steps: “Part of the spirit of our method is to choose steps of the appropriate “size”” [Plo81]. Following the evaluation of a complex expression in a transition system generated by term rewriting would be very tedious. A transition system that takes expressions immediately to their value might be preferable for some purposes, or we might want to see only the “major” steps. This can be arranged by making the kinds of transitions you want to see the conclusions of the rules, whilst the ones you don’t want to see become premises.

It is difficult to see how any of these advantages of structural operational semantics over plain term rewriting are applicable in the graph transformation case, especially when the goal is to formalise a behavioural language such as UML.

Each term has a root, or main-connective which classifies it, allowing the right rules to target the right terms. Graphs do not have a hierarchical structure like this. GOS rules, like ordinary graph transformation rules, are applied via a graph homomorphism called a match. To have the structural control of the transformation as SOS rules do, we would have to require this match be surjective, which would be an impractical limit on the state space: we want system states that are bigger than the left-hand-sides of graph transformation rules. In short, the structural benefits of structural operational semantics can not be extended to graphs, because graphs do not have that kind of structure.

What is the proper “size” for the transition steps in UML’s execution semantics? We suggest that since “all behavior in a modeled system is ultimately caused by actions executed by so-called “active” objects” [Obi07c §6.3.1], that each transition represent the parallel execution of one or more actions. Since actions are the atomic units of behavior, they should be represented by graph transformation rules, one rule for each of the 36 concrete actions. Thus no deduction system for transitions is needed except for parallel composition [EEP T06 §3.3-4].

The approach to the formalisation of UML exhibited in the DMM work originates before the UML actions, and the substantial revisions of UML 2.0. For example, in [CHM00], an event occurrence is represented by a transition between two system states. Such a transition is the premise for the rule which makes a state-machine transition. Thus the occurrence and its effect happen at the same
time. This is made possible by the use of GOS, but is it desirable? As we noted in Section 5.3, event occurrences and operation calls are not transitions, but persistent entities that reside in “pools” and are later “consumed”.

DMM defines the semantic domain by extending the metamodel. Thus instances of this graph contain both the model proper and the system state, as we required in Criterion 4. Small examples are given in [EHHS00] and [ESW07]. In the earlier paper, a fragment of the UML 1.3 metamodel describing objects and state machines is extended with an association current between metaclasses Object and State, so that at run-time, instances of object can be linked to their current state. The later paper [ESW07] follows the same pattern by extending a fragment of the activity part of the metamodel with runtime elements Offer, Token and ActivityExecution.

The metaclass Object was included in the earlier UML metamodels. This was later recognised to be a conceptual error, and it has been removed in UML 2.x. It was an error because objects are part of the world being described, but classes are part of the description of that world. That is, classes are in the syntax whereas objects are in the semantic domain. For this reason, Object has been replaced in UML 2 by InstanceSpecification. Object and BehaviorExecution do appear in the “informal” semantic domain model of [Obj07, §13.1]. This model does not address Activities, but if it were to be extended, the Offer, Token and ActivityExecution classes would fit comfortably.

Dynamic metamodelling deliberately merges the models of the language and the semantic domain. Defining the semantic domain by extending the metamodel has the happy effect of putting the model in each system state, although the hierarchy does not continue. Unhappily the distinction between model and system state is lost, because they are all described by elements of the same metamodel. How do we know that models may contain Classes and Activities, but not Objects and ActivityExecutions? We guess that the current metamodel, a subset of the extended one, would be designated as the syntax model. When the concerns of syntax and semantics are separate, we have the possibility of using different languages to talk about the same kinds of systems, or using the familiar language to talk about a new kind of system. This would be difficult under DMM. In Section 8.1 we outline a way that syntax and semantics can be defined separately, then merged to create a single type-graph in the style of dynamic metamodelling.

Dynamic metamodelling has found the right general approach to defining UML’s semantics, but has erred in some of the details. In particular, UML’s action and causation mechanism is not captured, probably because it was formulated after DMM. It is therefore not completely faithful to the existing language, as required by Criterion 4 but our own work in Chapter 5 indicates that a faithful version is possible. The approach is understandable (Criterion 6), it is based on uniform use of simple mathematics, but uses existing UML notation to express it. Because a system state is an instance of the combined syntax/semantic domain metamodel, the user-model is available in the run-time state, satisfying a single-level version of our Criterion 4.
Other Graph Transformation Approaches

A different approach to defining UML's semantics using graph transformation systems originates at the University of Bremen [ZHG05, GZK02]. Its most mature presentation so far is to be found in the 2005 paper of Paul Ziemann, Karsten Hölscher and Martin Gogolla [ZHG05]. This paper describes integrated formal semantics for an impressive range of UML's diagram types. They also describe a partially completed interactive model execution tool based on these semantics.

The dynamic metamodelling approach embraces the UML metamodel, extending it to yield a type graph for the graph transformation system. The Bremen group on the other hand, note that the definition of UML's abstract syntax by a metamodel appears circular, and subsequently ignore it [ZHG05]. The type graph for their graph transformation systems contains a form of the class diagram fragment of the metamodel, including classes such as Class, Association and Operation, along with their run-time counterparts Object, Link and Process. Thus, the class diagram part of the metamodel is present in the system state. However the other diagram types are handled in a variety of ad-hoc ways.

The meanings attributed to the diagrams is not derived from the official definitions of the model elements they represent, but rather “relies on a number of assumptions on how the diagrams could be used in practise and integrated in a useful way” [ZHG05]. So for example, they have chosen to use collaboration diagrams to specify the sequence of operation calls used to implement another operation. Use-cases are also considered to be operation declarations. Thus what is presented is not really a candidate for the formal semantics of UML, but rather formal semantics for one particular style of UML use. This does not prevent us from considering their general approach as a way of clarifying the existing semantics.

The most significant difference between this approach and dynamic metamodelling is that here, each model yields a different graph transformation system. Many of the graph transformation rules are derived from the model’s diagrams. Hausmann says that this would make UML's dynamics more complicated to understand and implement [Hau05, Page 31]. More importantly, in the official definition the UML Actions and run-time control and causation apparatus are described in a general way, once and for all. A formal definition should preserve this, if feasible. We believe that the dynamic metamodelling work discussed above, and our own outline of a graph transformation system show that it is feasible.

Gogolla, Favre and Büttner also have a paper proposing graphs with special typing edges, allowing them to “squeeze M0, M1, M2 and M3 into a single object diagram” [GFB05]. Their aim is to give precise meaning to some metamodel properties that are discussed in the literature, such as potency and strictness. These special edges can join pairs of nodes as usual, but can also join pairs of edges. Interestingly, one of the properties they formulate, “strongly well-typed”, means that these edges are a graph homomorphism.

Another translation of a UML fragment into graph transformation systems is presented in [BCG04]. In this paper Baldan, Corradini and Gadduci use the
class diagram as a type graph, in this case a hypergraph. Each node has at most one outgoing edge with several labelled “tentacles”. There is no discussion of what difference this choice makes, or why it was made. Atomic activities are defined as graph transformation rules using collaboration diagrams, as pioneered in [EHHS00]. They then study an activity diagram which specifies the sequencing of several such atomic activities. Given an initial state for their example system, they prove that the atomic activities can only be executed in the order the diagram shows. That is, rather than treating the activity diagram as prescriptive, they consider it descriptive and requiring verification.

The much more significant contribution of this paper is the use of a logic, evaluated over the graph transformation system, to express and verify system properties. The logic used comes from the author’s earlier work on verification of graph transformation systems (but see [BCKL07], for more mature work). The static part of the logic is monadic second order, which means that it has variables for edges and for sets of edges. Formulae of this logic are the propositional atoms of a modal \(\mu\)-calculus, which allows the definition of the usual modal and temporal operators. They use this logic to express several useful desired properties of their example model, but find that it can not express others. Because the temporal operators can not appear within the scope of a quantifier, it is not possible to say anything about the same object at two different points in time. They give an example “All boarded passengers arrive at [their] destination.” The language is extended to allow this kind of statement, but the authors did not know whether the verification techniques could be extended to apply to this richer language. Further work on this question appears in [BCKL07].

We were surprised by the treatment in [BCG04] of an activity diagram as a descriptive rather than a prescriptive diagram. A similar surprise is to be found in Aliki Tsiolakis and Harnut Ehrig’s work [TE00]. Their aim is to check consistency between class and sequence diagrams. To do so, they translate the class diagram into a type graph, and translate sequence diagrams into a controlled graph grammar. That is, a collection of graph transformation rules along with a control condition describing their order of application. The association end multiplicities of the class diagram are also captured using graphical constraints, an idea due to Heckel and Wagner [HW95]. Techniques are developed to check whether the sequence can be executed in a given state, and whether the resulting system states will be consistent with the class diagram. This is achieved by checks involving the rules, and in some cases an initial state, without actually generating a sequence of system states. Unfortunately, the properties that are checked are not useful with respect to UML as currently defined. For example, they insist that if an object sends a message to another, then there must be a link between those two objects. Another example is that their multiplicity constraint check does not account for any of the difficulties we discussed in Chapter 4. However, techniques analogous to those presented would be of practical benefit, if they were designed to test relevant properties of systems specified with UML as it is defined.

Model transformation is one of the key ideas of model driven development. If
models are graphs, then it is not surprising that the well studied theory of graph
transformation has also been applied to model transformation [VF02, GGL05,
EEE07].

Graph transformation is a very suitable way of defining UML, as we have been
arguing throughout this work. The formalisations of UML using graph transforma-
tions that have been offered thus far are not sufficiently faithful nor extensive to
be convincing demonstrations, but dynamic metamodelling could become the basis
of a definition that satisfies all our criteria. We prefer a single graph transforma-
tion system “interpreter” approach, since this is more sympathetic to the existing
informal definition than a translation approach. The other main question is what
should serve as the type graph, the user model or a semantic domain model? The
answer, we believe is both. We show how this can be achieved in Section 8.3.1. Sev-
eral contributions as well as our own work have shown that GTS can be used to
define both the abstract syntax and semantic domain of UML. We have seen that
model-checking and other techniques can be applied to test properties of systems
defined in this way. Existing UML diagrams have been used in a natural way to
define graph transformations systems [EHHS00]. Therefore, if this subset of UML
is easy to understand, at least intuitively, then so are graph transformation systems.
Reflection can be supported by instantiation edges. Flexibility has not been explic-
itly addressed by any work we have seen, but one could clearly vary the semantic
domain model or graph transformation rules as required. The question would then
be, how much of the underlying theory and tool support could be retained?

What remains to be done is to work out the details of a definition that it faith-
fully captures and clarifies the language.

7.3 Logics

We see three distinct roles that a logic could play in defining UML. First, as a se-
monic foundation or replacement for OCL. Second, to replace the current flawed
treatment of Interactions or sequence diagrams (see Section 5.4). Thirdly, to fa-
cilitate tool support, in particular, deductive verification and model checking. The
contributions we study in this section do not always have goals that align with this
view, but this is what we are looking for. We begin by examining several contri-
butions that verify UML models using logical model checking. This leads us into
yet another (see Sections 1.3 and 5.3) discussion of description verses prescription
and its consequences for verification tasks. We also consider the expressiveness of
the logics used. An example of deductive verification follows, and then we look at
several papers on foundations for OCL and dynamic logic for modelling language
semantics. For a brief introduction to formal logics including dynamic logic, see
Section 6.1.

Some of the work discussed in Section 7.2 on graph transformation employed
model checking [Ren03, Hau05, ESW07]. Model Checking [MOSS99] takes a
logical formula and a structure, and determines whether or not the structure satisfies
the formula. That is, using the terminology we developed in Section 1.3 whether or not the structure is a logical model of the formula. The structure is typically a model (in the non-technical sense) of some system, and the formula states a property we would like the system to have. The structure is usually graph-like, with nodes representing system states and edges representing transitions. Modal and temporal logics are typically used to express the desired properties. It seems natural to verify UML models by deriving a structure from the prescriptive parts (StateMachines, Activities), and formulae from the descriptive parts (Interactions).

This is the approach taken in [CF05], [XLB01] and [MC01]. In the first paper [CF05], Couzinier and Féraud translate example sequence and state machine diagrams into TLA+. This extension of Lamport’s temporal logic of actions is the input language for the model checker TLC. The second paper [XLB01], by Xie Levin and Browne, develops an automated translation from a UML subset with classes and state machines into the specification language of the COSPAN model checker. It also develops an OCL like language for expressing the desired system properties. Model transformations are applied to the UML model to reduce the state space of the resulting structure. In the third paper [MC01], McUmber and Cheng set up a general scheme for translating UML into formal languages using model transformation. This scheme, makes use of a notion of homomorphism for models. If we incorporate the insight that models map homomorphically into their metamodels, the scheme can be depicted as in Figure 7.6. Given a source model $s$ instantiating a metamodel $S$, with $S$ in turn mapped homomorphically into a target metamodel $T$, the task is to find an instance $t$ of $T$. The dashed line is the required instance translation, which is partly specified by rules. The main specific translation presented in this paper is into the input language of the Spin model checker.

Each of these contributions employ a model checker for a variant of linear temporal logic (LTL). The evolution of non-deterministic discrete processes, like a UML system, takes the form of a tree. Each node is a state, and each outgoing edge leads to one possible successor state. There are logics, such as computational tree logic (CTL) which directly evaluate formulae over such a structure, but LTL evaluates formulae separately over each path of the tree, and considers it true if it is satisfied by every path. This results in a loss of important expressiveness, as the following example from [MOSS99] shows. Figure 7.7 shows finite state automata representing two vending machines. The one on the left is preferable.
because after inserting a coin, the user can choose whether she wants coffee or tea, whereas in the right one, the machine makes the choice as soon as the coin is entered. However, the paths (or words) of these automata are the same, and so they are indistinguishable to LTL.

Is this expressive weakness a problem for the purpose of verifying sequence diagrams? In other words, could a sequence diagram distinguish between the two automata of Figure 7.7? Consider an Actor lifeline sending a coin message, then a tea message to some object, as shown in Figure 7.8. The object enters an activation on receipt of each message, indicating acceptance. As we discussed in Section 5.4 and in Chapter 6, the meaning of a sequence diagram is not really complete until we add some modality. The simplest possibility is to assert that this sequence of interactions can happen. This modality makes both automata satisfy the sequence. Another reasonable interpretation would be that once the first message has been sent, it is always possible for the second to be sent (and accepted). That is, we take the sequence diagram to mean that when the user inserts a coin, he is always able to then order tea. The official definition [Obj07c] does not specify the meaning to this level of detail. Formulated in dynamic logic, this is $[\text{coin}] (\langle \text{tea} \rangle \top)$. We may also insist that it is possible to insert a coin in the first place, yielding

$$\langle \text{coin} \rangle \top \land [\text{coin}] (\langle \text{tea} \rangle \top).$$

(7.1)

This suggests a general scheme, where a sequence diagram containing $n$ linearly ordered messages means $\langle m_1 \rangle \top \land [m_1] [m_2] \top \land \ldots \land [m_1] \ldots [m_{n-1}] [m_n] \top$. However, consider formula (7.1) as a potential meaning of the example sequence diagram. In CTL, the structure (vending machine) on the left satisfies this but the one on the right does not. The left is equivalent to the right under LTL, so neither satisfy the formula in that logic. That is, an LTL-based model checker would incorrectly reject a satisfactory vending machine.

Of course this does not mean that LTL based model checkers are useless. Reducing a tree to its set of branches is a kind of abstraction, which loses information. What is required is a systematic account of what abstractions are available and what
information they lose.

![Figure 7.8: Sequence Diagram](image)

The three papers that inspired this digression offer a way of detecting errors by translating parts of a model into the language of a model checking tool. Even if a CTL based tool were used, this translation necessarily involves some abstraction, and perhaps distortion because of the differences between the semantic domains of the logic and UML. Ideally, we should have an understanding of the faithfulness of such translations and abstractions. We made the point before in Section [1.3] when discussing the relationship between a target system and its model. When we further abstract a model for tool analysis, it would be good to know what errors would be detected and which would be missed, and what false error reports could we expect. An excellent starting point for this would be a fully expressive language defined over a precisely defined semantic domain for UML.

The first plausible demonstration of deductive verification of UML models is given by Arons, Hooman, Kugler, Pnueli and van der Zwaag, in [TAKPvdZ04]. The semantics are not described in this paper, but are derived from those of [DJPV03]. That paper gives formal semantics to a small executable subset of UML intended for real-time applications, using Pnueli’s “symbolic transition systems.” Much of the considerable complexity of that work comes from the need to model hard real-time systems, which makes us wonder whether the general modelling community might get by with something simpler. The abstract syntax of the official definition is ignored, and a traditional formal syntax is given for the selected UML subset. The later deductive verification work uses a temporal logic embedded into the higher order logic of the PVS interactive theorem prover. A model given by a class diagram and state machine diagrams with some actions, is automatically translated from .xmi form into PVS sources. Issues of consistency are deliberately avoided, since deductive verification of liveness properties (system does not get stuck) and safety properties (bad things do not happen) are challenging enough at this stage. Several strong assumptions are made about the execution semantics, which are not present in the official definition. The liveness proofs depend on the
fairness assumption (every active object gets to act), which probably should be in the definition. The “run to completion” assumptions seem rather stronger than those in the definition. Deductive verification is not required for most applications of UML, but supporting formal proof demonstrates that a definition is precise and unambiguous, which we have demanded in Criterion 2.

This formalisation uses a language with both temporal and higher order features, so it is not subject to many of the limitations we have identified for other approaches. Most of our criteria are not addressed by this work however. Most urgently, we need techniques to check consistency, and we need the meaning of models to be understood by non-technical modellers and end users.

The Object Constraint Language (OCL) [Obj06b] is very much like the languages of traditional symbolic logic, and at least two groups have attempted to make it precise by translating it into well understood systems of logic, intending to enable theorem proving about models. Brucker and Wolff [BW02] use higher order logic (HOL) as implemented in the generic interactive theorem prover Isabelle [PN]. Beckert, Keller and Schmitt [BKS02] use first order logic. OCL 2.0 has a third truth value “undefined” and allows collections of collections, so first order logic will probably not suffice to formally define it. Neither group make use of the OCL metamodel in their translations. Schmitt and Beckert’s group offer different, equivalent translations optimised for readability or for automated theorem proving respectively. With a foundation as suggested in these works, OCL itself could be the target formal language for a model transformation defining the semantics of UML. This would probably require additions to the current limited temporal operators of OCL though.

Semantics in this form would be more likely to cultivate widespread precise understanding of UML models. The ultimate translation from OCL into symbolic logic might not be popular reading, but it could facilitate access to the huge body of work on theorem proving and constraint solving, providing tools for consistency checking and related tasks. The decision to include higher order concepts and a third truth value in OCL 2.0 might prevent effective use of most of that work though.

The OCL formalisation of Beckert and Schmitt [BKS02] is used in their KeY project [ABB05]. This is a tool for the deductive verification of Java-Card programs using a specialised dynamic logic [Bec01]. This logic is implemented in a generic theorem prover integrated with the Together modelling tool [TC], and thus provides a practical platform integrating UML modelling and formal methods. Although this work is not aimed at improving the definition of UML, it is instructive. The deductive rules symbolically execute the Java-Card program, and thus give a clear and precise account of the language semantics. The rules could even provide educational interactive animations of the language.

Unlike Java-Card, UML is non-deterministic and has no main procedure, but it is conceivable that one could develop such a dynamic logic specifically for UML. The shortcomings of standard dynamic logic which we discussed in Section 6.4 could thus be overcome. The logic would have rules for each of the UML actions.

152
As suggested in Section 7.2, the semantic domain could be a graph transformation system, with its transitions labelled by the graph transformation rules that generated it. These rules would implement the UML actions. This would define model dynamics, and the meaning of descriptive dynamic model elements such as Interactions could be expressed by interpretation functions which yield formulae of the dynamic logic language. It would also enable deductive verification of UML models without the semantic distortions of translation.

Logic is not the only means of defining semantics for OCL. The work of Richters and Gogolla [RG01] is included in the official OCL definition [Obj06b] as an “informative” annex. This defines the syntax and semantics in the direct mathematical style typical of formal logics, only more complex. Cengarle and Knapp [CK03] give a type inference system and small step structural operational semantics (SOS), and prove type safety. They also give a big step SOS and denotational semantics, applying these to a proof about the expressiveness of OCL.

OCL is intended for writing constraints about UML models. It therefore must have the same semantic domain as UML, otherwise, it is simply off topic. We have already suggested that a precise version of OCL could provide a foundation for UML. UML models could be translated into formulae of this language. Authors proposing semantics for OCL must either account for an interpretation of UML into the same semantic domain, or show that a translation is possible.

Executable UML [MB02] is Stephen Mellor’s development method. Its predecessor, the Shlaer-Mellor method [SM92] is studied by Wieginga and Saake in [WS96]. Another shorter paper [WB97] studies a subset of UML 1.1, but the work is so similar that we focus on the earlier contribution. The visual modelling language of the Shlaer-Mellor method is given formal semantics using their own language LCM, which is based on dynamic logic. This logic is first order, but the programs are not explicit assignments, but rather unanalysed atomic programs like those of PDL (see Section 6.1). To ensure that these programs have the desired effect, axioms are introduced. Guard axioms have the form \( \langle \alpha \rangle \top \rightarrow \varphi \), which means \( \alpha \) can only execute if \( \varphi \) is true. Effect axioms have the form \( \varphi \rightarrow [\alpha] \psi \), which means if \( \varphi \) is true, then after \( \alpha \) executes, \( \psi \) must be true. Other assumptions not expressible as axioms are also required.

Shlaer-Mellor is subjected to a “methodological analysis” resulting in many suggested revisions, some of which appear in Mellor’s newer method, Executable UML [MB02]. Some of the revisions though, seem too extreme.

Our methodological analysis has led to a system transaction concept that is simple to formalize: Each system transaction is a synchronous execution of a non-empty finite set of local object transactions. Because each object transaction has a local effect only, this composition is harmless and we can simply conjoin the local effects.

Thus tracing a timeline of object interactions, as described by a sequence diagram, would be impossible. Wieiringa’s goal is requirements engineering, which is not
concerned with the internal behaviour of a system, only its interaction with its
environment. Model driven development asks more than this of its models. We
require the ability to fully describe a system, including its internal operations.

Our own work in Chapter 6 and [O'K06a] also uses standard dynamic logic
[HK100] to give semantics to a UML subset.

Most of the contributions we have considered translate some part of UML or
OCL into a known formal logic. There is always some encoding or abstraction
involved, but this is never seriously studied. The situation would be made much
clearer if the semantic domain of UML were defined, and a readable but rigorous
formal language defined over this domain, expressive enough to render anything
that a designer might want to say about their system. The language would of course
be undecidable, but would stand as a reference point from which fragments and ab-
stractions could be assessed. Such a language could also be used for constraints,
and to define the meaning of difficult, descriptive model elements such as Inter-
actions. The beginnings of such a language may be present in work on graphical
logics [BCKL07, Ren03, HW95]. We shall return to this idea in the conclusion,
Section 8.2.

### 7.4 Other Techniques

This section surveys a range of “other” approaches to formal semantics for UML,
and related work.

**Algebra, Coalgebra and Categories**

A category is pretty much like a graph, indeed, the definition of graph that we
showed in Figure 3.5 (on Page 36), which was taken from [LS86], is extended
later in the same page of that book to define categories. The directed edges are
usually called *arrows*, and the nodes *objects*, not be confused with the idea from
object oriented programming and modelling. A category also has a composition
operation allowing arrows to be joined together, and a special identity arrow for
each object. Category homomorphisms are called *functors* and there is a kind of
arrow between parallel functors called a *natural transformation*. These ideas have
been found useful in mathematics because they turn out to be very general. Lots of
structures can be seen as categories, as can collections of all structures of a certain
type with the homomorphisms as arrows between them. Relationships between
these structures can often be expressed as functors, and preservation properties of
the functors allow results to be transferred from one field to another. Categories
have also become surprisingly popular in theoretical computer science.

Zinovy Diskin [Dis98, Dis01, Dis03] argues that category theory is the correct
formal foundation for software and business modelling in general, and for UML
in particular. Software engineering is like mathematics in that it deals with sys-
tems of concepts, and seeks structure underlying these systems. Therefore meta-
mathematical tools for organising these mathematical systems of concepts ought to be applicable to software engineering too. Metamathematics is usually done with logic, but a rival approach uses category theory. This is the more suitable approach for software engineering, argues Diskin. He reviews techniques used to combat complexity in software engineering and compares them with parallel techniques in category theory. Encapsulation of objects and modules is compared to the category theoretic practice of defining structures in terms of the arrows to and from them, rather than directly prescribing their internal structure. Variable sets are cited as category theory’s notion of object identity. Visual notations, such as those of UML, are seen by Diskin as sugared versions of his notion of sketch (several ideas compete for this name in category theory). Diskin’s sketches are graphs, with arrow ends annotated for injectivity, totality and so on, and with certain subgraphs labelled to indicate category theoretic diagram properties such as pullback and inverse. This is a kind of language for describing categories, which he considers more suitable for software work than logic. Logic, he points out, is textual, whereas software engineering, at least at the analysis and design stages, is graphical. Classes and such software ideas specify sets, whereas logic tends to make statements about individuals. Diskin is thinking of first order logic, the preferred logic of mathematical foundations. Finally, category theory has a rich system of “multi-level specification” in that its structuring ideas can be applied to its own structures: categories of categories, categories of functors and so on.

Diskin does not make an explicit proposal for a more precise definition of UML, but we presume he would like to see a document with his sketch diagrams in place of the confusing English semantics. His treatment of UML dynamics is limited to suggesting that classes should denote variable sets and state machines denote coalgebras.

Diskin’s categorical sketches could be used to give a precise and faithful definition of UML’s static semantics. His treatment of UML diagrams as variant notations of his sketches gives a clear interpretation to them, however, it completely bypasses the metamodel, which is essential for model transformation. Model consistency would reduce to the question of whether there was a category satisfying the sketches represented by the model diagrams. Model refinement would probably mean the existence of a functor between such categories. The important ideas could be made quite precise, but how many people would understand them? Although graph transformation makes use of category theoretic ideas, the simple nodes and arrows can help to clarify ambiguous models. Non-mathematical people will not benefit by being shown the sketch form of a disputed UML diagram. The only software tools we are aware of for analysis of (finite) categories is [RB88].

A more substantial account of object dynamics in terms of category theory can be found in the work of Joseph Goguen [Gog92]. Objects are defined in terms of possible observations. In particular, the open sets of a topological space are taken to be the possible points of view, or places and times of possible observations, and the object itself is, for each of these open sets, a set of functions into the product of the object’s attribute domains. When such a structure admits a “fusion” of ob-
servations that agree pairwise, it is called a sheaf \cite{MM92}. The topological space need not be discrete, so continuous phenomena can be modelled in this scheme. Inheritance is captured by homomorphisms between objects, and possible interactions between objects are defined by them both inheriting from a third, interaction object. Thus a system is specified as a category theoretic diagram showing the objects and their inheritance relationships. The category theoretic limit over this diagram yields an object (sheaf) whose behavior is the combined behavior of the system. Systems of systems may be composed as diagrams of diagrams, and the combined behavior in this case is a colimit. Examples are given of semantics for a concurrent programming language, deadlock analysis and security in terms of “non-interference”. What Goguen proposes is a scheme of modelling using category theory, a competitor for UML rather than a foundation for it. There is no modelling language, other than the diagrams and mathematical symbols of the category theorists. No specific suggestions are made about the use to be made of the models in the development process, or tools to analyse them. The notion of system developed is elegant and extremely general. It seems likely that using such neat and powerful concepts could lead to better quality designs. But mathematical ideas have never been popular among software developers. In practice, a clarified form of UML has much more chance of improving software quality than an attempt to introduce category theory.

In Chapter \ref{Chapter6} we mentioned variants of dynamic logic which allow functions to be updated. The same idea has been applied directly to multi-sorted algebra in what was originally called “evolving algebras” \cite{Gur95}, but are now known as abstract state machines (ASM). A concise description is given in \cite{BCR04}, which we shall discuss presently.

ASMs are transition systems, their states are multi-sorted first-order structures, i.e. sets with relations and functions, where for technical convenience relations are considered as characteristic boolean-valued functions. The transition relation is specified by rules describing the modification of the functions from one state to the next, namely in the form of guarded updates (‘rules’)

\begin{verbatim}
if Condition then Updates
\end{verbatim}

where $Updates$ is a set of function updates $f(t_1, \ldots, t_n) := t$, which are simultaneously executed when $Condition$ is true.

ASM specifications read like programs with a little algebraic syntax added. They thus provide a formal yet approachable way of specifying both structure and dynamics.

The short paper \cite{BCR04} by Börger, Cavarra and Riccobene summarises and discusses a series of their previous papers on formalising UML state machines using ASMs. They pay close attention to the official definition (version 1.4) and
treat hierarchical states, capturing ideas about active states and event handling as ASM structures and rules. In the process, they detect several difficult and unclear points in the official definition.

Gurevich is among the authors of another piece on ASMs for UML formalisation [CGHS00]. State machines again receive most of the attention, but class diagrams are also considered. A translation is employed from ASMs to the input language of the SMV model checker, enabling verification of a model against specifications written in CTL. This is demonstrated with a simple example where a non-obvious modelling error is detected by the tool. In contrast with the previous work, the official definition is not mentioned or cited. The standard of written English is not high and large sections do not seem relevant.

Ileana Ober's ASM formalisation [Obe03] is a kind of dual to these other contributions, in that it treats a significant fragment of UML but excludes state machines. Instead, this work focusses on the UML actions, each of which is coded as an ASM macro. The official definition of the time (1.4) is taken seriously, and the definitions begin with an automatic translation of the metamodel into ASM definitions. Thus our reflectivity criterion is satisfied: the model is present in the run-time system state. The OCL and natural language well-formedness constraints have also been coded, providing a syntax check. Run-time specific concepts such as signals, event queues and operation method look-up are also defined after careful consideration of the explanations in the official definition document. Where the official definition is silent, ambiguous or confused, explicit choices are made. Ober does not consider her formalisation as an aid to understanding UML, rather as a means to facilitate tool support and consistent results. She suggests that the ASM definitions should be hidden from the modeller. There is also a distinction between repository and run-time semantics. The metamodel is present only as ASM sorts and functions, then the model populates this. Much less space is given in the paper to describing the actual objects, though the authors PhD thesis works through the proposal in detail. The object’s instantiation of the model elements is most likely represented by a function, which would need to be set-valued to enable multiple-instantiation (Section 3.3).

We should also mention that a meta-model form of the ASM language has been produced [RS04]. This would facilitate model transformations between ASM and UML.

Between them, these ASM related contributions demonstrate that a faithful formalisation of UML's main ideas is possible. In particular, concurrency is built-in. Although somewhat mathematical, abstract state machine notation would be approachable by reasonably educated tool-makers and modellers, allowing a clearer understanding of the semantics to propagate. The modularity of ASM specifications should assist in providing the required flexibility. There are tools available for executing and analysing abstract state machines, including the Spec Explorer produced by Microsoft [Mic].

Algebraic specification [Wir90] extended with “generalised labelled transition systems” is used by Gianna Reggio, Maura Cerioli and Egidio Astesiano to for-
malise parts of UML in [RCA01] and earlier papers by the same authors. They do this by translating UML diagrams into the language Casl-LTL, though they emphasise that the particular language is immaterial. This work explicitly aims for a way of giving useful formal semantics to the whole of UML, and as the title suggests, they take seriously the idea that the different diagrams combine to specify a single system.

The authors note that it is not enough to define semantics for one or two diagram types. Rather, a semantic framework is required such that any collection of legal diagrams can be considered as a specification for a single system. They refer to this as UML’s “multiview” approach. This is an important insight which enables us to recognise the limited value of much of the existing work.

However, their work fails to align itself with the metamodeling technique used to define UML.

we propose a general schema of what a semantics for UML should provide from a logical viewpoint, combining the local view of a single diagram with the general view of an overall UML model.

The subtle error here is that a diagram is a piece of concrete syntax, whereas the semantics is to be applied to the abstract syntax. A collection of concrete diagrams specifies a single abstract syntactic unit, a population of the metamodel, a model. In fact, very little of the work on precise UML semantics takes into account that the syntax is actually the instances of another model.

The multiview nature of UML then, must be accounted for not in the semantics, but in the notational conventions by which the abstract syntax is read from the concrete syntax. These conventions have not been given much attention in the literature, but it is not clear that they are free from difficulties.

The authors note the expressive demands made by UML’s dynamic diagrams.

It is worth noting that to state the behavioural axioms we need some temporal logic combinators available in Casl-LTL that we have no space to illustrate here. The expressive power of such temporal logic will also be crucial for the translation of sequence diagrams... Indeed, Z and its derivatives would face similar difficulties. A later paper [AK02] by Astesiano and Reggio studies UML consistency from their algebraic point of view, and also uses a metamodel to describe the formal language being used.

Perhaps the earliest relevant work on algebraic specification for UML is [BC95], where Bourdeau and Cheng use Larch to formalise the OMT [RBP+91] predecessors of object diagrams.

The implicit suggestion in Gougen’s and Diskin’s work is that category theory could be used as a conceptual model in much the way relations are in the relational model. This would certainly make some powerful conceptual tools available, but powerful does not always entail useful. It would be interesting to see how these ultra-abstract ideas could be put to work in a practical setting. Much more in the
spirit of the present work, Ober’s ASM formalisation of UML [Obe03] seems to present a viable alternative to graph transformation.

Z and Object-Z

Z [Spi92] is a formal specification language which uses basic mathematics and logic notation to define a space of system states and transitions between them. It is probably the best known artifact of so-called “formal methods”. It aims to make specifications precise to enable formal reasoning about them. In this subsection, we survey a few of the many contributions that use Z, or its variant Object-Z, to formalise parts of UML.

The best way to introduce Z is to give a simple example, which we have taken from [Lig91]. This example describes the action of the “home” key in a text computer terminal with a cursor. First, a basic type KEY is declared.

\[ KEY \]

Then a couple of global constants.

\[
\begin{align*}
\text{numlines} & : \mathbb{N} \\
\text{numColumns} & : \mathbb{N} \\
1 \leq \text{numlines} \\
1 \leq \text{numColumns}
\end{align*}
\]

The declarations go above the line, the conditions below. Collections of declarations and conditions can be given names and reused.

\[
\begin{array}{c}
\text{Cursor} \\
\text{line} : \mathbb{N} \\
\text{column} : \mathbb{N} \\
\text{line} \in 1..\text{numLines} \\
\text{column} \in 1..\text{numColumns}
\end{array}
\]

Many standard mathematical symbols and keywords are defined. The following names 6 distinct elements of the KEY type.

\[
\text{left, right, up, down, home, return} : \text{KEY}
\]

\[
\text{disjoint}\{ \text{left}, \text{right}, \text{up}, \text{down}, \text{home}, \text{return} \}
\]

Now we have the context needed to define the action of the home key. The Cursor schema is reused with the $\Delta$ convention, which means that all the variables of that schema are included here, as well as primed copies of them. So, the following schema will have line and line’ among its variables. These primed variables refer to the successor state. The $\Delta$ schema also contains primed copies of the conditions,
so $\Delta \text{Cursor}$ contains $\text{line}' \in 1..\text{numLines}$. The variable $\text{key}'$ uses the convention that input variables have a ? suffix. This has no semantic significance, its just a hint to the reader. The conditions of the following schema say that the key is home, and in the next state, the line and column are both 1.

<table>
<thead>
<tr>
<th>HomeKey</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta \text{Cursor}$</td>
</tr>
<tr>
<td>$\text{key}' : \text{KEY}$</td>
</tr>
<tr>
<td>$\text{key}' = \text{home}$</td>
</tr>
<tr>
<td>$\text{line}' = 1$</td>
</tr>
<tr>
<td>$\text{column}' = 1$</td>
</tr>
</tbody>
</table>

Later, after similar schemas have defined the action of the other 5 keys, the cursor control is defined by the disjunction (or) of the 6 schemas. Object-Z extends Z with some object oriented notions.

Kim, Carrington and Burger [KC00, KBC05] give explicit translations between Object-Z and class diagrams. In the earlier work, the syntax of both languages is expressed in Object-Z, and the translation is also defined there. A metamodel of Object-Z is provided for the benefit of modellers unfamiliar with this formal language. In the later work, the metamodels define the syntax, and the translation is defined using a dedicated model transformation language. Unfortunately, even this recent work only addresses a subset of the class diagram fragment of UML. The work aims to enable formal verification of UML models, but as yet we have no demonstration nor descriptions of specific techniques.

Model Driven Architecture [MM03] aims to enable the simultaneous use of many languages, each with syntax defined in MOF, by using model transformations between these languages. The real contribution of work such as [KBC05] is in recognising that formal languages can also participate in this way. Above, we discussed another which does this [MC01], and one which we have not discussed [VP02]. Definitive formal semantics could be provided by a Z (or Object-Z or ASM [RS04] or dynamic logic or . . . ) metamodel and UML to Z model transformation. This would enable tool integration, and provide insight into the formalism for the more advanced modellers. Attempts to directly translate diagrams into formal languages usually ignore the metamodel definition of the language, and thus violate Criterion 9.

Bruel and France [BF98] advocate an integration of UML and formal methods, in which a UML class diagram is translated into the formal specification language Z. The Z specification is then manually refined, adding details not expressible using class diagrams. The rules and guidelines for semi-automatic translation, they hope, will give insights for developing a more precise semantics for UML. Since Z is well supported by tools, the resulting Z schemas can be checked for consistency. The authors stop short of actually suggesting that UML semantics should be specified using Z.
Rasch and Wehrheim [RW03] also advocate integration of a formal language, in this case Object-Z, into the development process. The Object-Z specification manually derived from the class diagram also specifies the class operations. The class is further constrained by a protocol state-machine, which together with the Object-Z schema, is translated into Hoare’s system of Communicating Sequential Processes (CSP) [Hoa85]. The choice of CSP, which is even less readable than Z, seems to be motivated mostly by the availability of a model checker (see the Logic subsection above) which they aim to use for consistency testing. They consider several notions of consistency and study which of these are preserved under CSP notions of refinement. We are not convinced that the intended semantics of the UML fragment are captured by this translation. It is also not clear that the CSP notions of refinement are applicable. The authors claim that the Object-Z to CSP translation can be automated. However, Criterion 5 demands a flexible semantics, which can be adapted as required. Knowledge of both Z and CSP seems too much to ask of the people who will perform these semantic adaptations.

CSP is also used as the underlying semantics for another approach to UML consistency checking. Jochen Küster and coauthors aim to characterise model manipulations which preserve certain notions of model consistency [EKHG02, KS03]. This is in order to avoid costly rechecking every time a model is edited. Consistency for these workers depends on the application, and each way of using UML will need to supply its own “consistency conditions”. For example, [EKHG02] describes using two statecharts together with a protocol statechart. The appropriate consistency conditions in this case are freedom from deadlocks between the two statecharts, and their combined behavior conforming to the protocol statechart. This seems rather ad-hoc to us. Our approach to consistency is to define UML’s semantics, which then enables us to apply the standard notion of satisfiability from logic.

The only attempt we know of to formalise sequence diagrams using Z is due to Miao, Lui and Li [MLL02]. We are not convinced that the intended semantics are captured by this translation. There is no use of primed variables nor Δ schemas nor sequential composition, which one would expect in such a temporal specification. The example class, state machine and sequence diagrams are unrelated, and there is no explanation of how the intended semantics are supposed to work in the resulting Z schemas. We suspect that any combination of diagrams would yield a consistent translation. Even if it is sound, this work does not contribute to better understanding of UML.

As we discussed in Chapter 4, the association end annotation {unique} is the subject of a recent controversy [Obj, Issue #5977]. Dragan Milicev [Mil06, Mil07] proposes semantics which reconcile the apparently conflicting parts of the UML definition. These semantics concern associations, their ends and the read, create, and destroy link actions. In an appendix to the report, Milicev gives an example model to illustrate the controversy, and expresses his semantics for it in Z. This is intended merely as a precise statement of the proposal explained in the body of the paper. However, this is the most convincing example we have seen of using Z to
express dynamic aspects of UML. It is also a good example of why Z will never be widely used by developers: it is not easy to read.

**Plain Mathematics**

Rather than reformulate UML as some specific kind of mathematical structure, such as graphs, some workers simply describe aspects of UML in terms of the most basic mathematical ideas: sets, relations and functions.

Thomas Baar takes this approach [Baa03] to define a MOF-like language in order to remove the circularity from the definition of UML's abstract syntax.

Broy, Cengarle and Rumpe propose a semantic domain for UML, which they call a “system model” [MB06, BCR07]. The semantic mapping from the syntax into this domain remains to be defined. The first part [MB06] describes the data store, which is the static parts of system states. This consists of a set of object identifiers, and a function assigning values to storage locations. Various types and type constructors are introduced, including record types which are used for object attributes. Functions are declared for various tasks such as yielding the carrier set of a given type, obtaining the value of a given location from a given state and so on. Interestingly, the authors decline to actually define associations, preferring to declare a function for retrieving the links of a given association which users of the semantic domain can define as they choose. This is because “even after quite a number of years of studying OO formalisations, there is so far not really a satisfying approach describing all variants of association implementations.”

The second part to this work [BCR07] deals with control and scheduling, which it defines in terms of threads. Early in the report, the authors apologise for a lack of elegance, “however, this lack of elegance accurately covers the lack of elegance in distributed object-oriented systems, where method calls, events and threads of activity are orthogonal concepts that can be mixed in various ways.” They go on to imply that the concrete benefit of elegance is ease of understanding. This is an important insight. We identified understandability as a key criterion, but its not easy to justify an assessment of understandability. It is often clearer whether or not something is elegant.

Great care is taken in this work to maintain UML’s distinction between operations and the methods which implement those operations. Again, various functions are declared to obtain parameter names and types, owning classes and so on. Threads execute methods, and maintain a stack of calls in the traditional way. They then go on to treat events (event occurrences) and messages (signal instances).

This work has the flavour of an implementation, using sets and functions instead of java or lisp. It would be straightforward to code it up, but as the authors observe, it offers little insight into the language. No other work we have seen considers the method resolution and control aspects of UML so carefully, but does it really need to get so complicated? The graph transformation approaches do not explicitly deal with threads, nor for that matter does the official UML definition. If an action is enabled, it is eligible to be executed. We would hope that the action being
enabled could be made to coincide with the applicability of the rule which executes that action. Threads, if we must discuss them, arise as a by-product of one action execution enabling others. Concurrency does not seem to be such a complicated issue in the graph transformation view either. Under certain conditions, rules can be applied simultaneously.

7.5 Comparison and Conclusion

Although it has been convenient to gather our discussions of these contributions according to the language or structure they use, we have also discovered several other dimensions by which they can be categorised.

First, there is the compiler/interpreter distinction pointed out by Jan Hausmann \[Hau05\]. His dynamic metamodelling with graphs and Ileana Ober’s work with abstract state machines \[Obe03\] and the system model of Broy, Cengarle and Rumpe \[MB06, BCR07\] provide a UML interpreter, whereas most other work translates each given model into an expression of some formal language. There is, we suspect, no theoretical significance to this distinction, both are just practical methods for defining one system of concepts in terms of another. For example, it is conceivable that exactly the same machine instruction executions could result from executing a program in an interpreter and executing a compiled form of the program. The difference is a practical one, and there are two kinds of relevant consequences. It may be easier in general to understand a language by having its ideas and execution mechanism explained directly once and for all, rather than having to consider each model as an encoded form of something else. Of course translation could provide an understandable definition if the target system is “conceptually close” to UML and the translation is simple and obvious.

This idea of “conceptual distance” is the second dimension of categorisation we wish to consider. We frighten the reader with quotation marks because we have yet to see a convincing account of how conceptual distance could be measured. Nevertheless, there seems to be an important idea there. The complex arrangements of stacks and threads in \[BCR07\] are a long way from the basic mathematical tools they employ. The authors lament their creation’s lack of elegance. Elegance, we suspect is a surprising lack of conceptual distance, a short-cut through ideas. Perhaps there is no short-cut through the thicket of ideas presented by UML. However, we see some reason for hope in our investigations. We have seen that the basic ideas of UML’s statics, objects and their properties, are neatly captured by the idea of a graph. An object diagram’s instantiation of its class diagram is a graph homomorphism. The fundamental building blocks of UML’s dynamics, the Actions, are well suited to employment as graph transformation rules. It appears that UML is conceptually close to graph transformation systems.

Application of graph ideas to formalisation of UML 2 has not been developed to the level of detail present in \[BCR07\] or \[Obe03\]. Perhaps the apparent elegance will not survive. If UML is fundamentally inelegant, and thus incomprehensible,
perhaps we should relinquish our criterion 0 (faithfulness to the official definition, see Page 12) and work instead towards a similar language which admits truly unifying abstractions. In this case, graphs would be the obvious place to start. Perhaps another candidate is Gougen’s sheaves [Goe92]? Theoretical computer scientists of a topological bent would agree that it is very elegant stuff, but UML is for the working software developer. If mathematics is to be used, it should at least be limited to non-specialist undergraduate level.

The final dimension which differentiates the contributions is tool support and computational complexity. The range of available tools is much more easily altered than the nature of the human mind. It therefore seems wise to choose a means of defining UML which is understandable, and only then consider tool support. UML is very expressive and its state space is certainly infinite, so automated theorem proving and model checking the full language is not likely. We have seen many ad-hoc translations which build in abstractions to achieve a decidable problem for known tools. We advocate instead making the whole language precise so that tool enabling abstractions can be properly understood along with the reliability of their results. The most appropriate starting point for this is a quantified temporal graph logic like [BCKL07].

In conclusion, it seems fair to say that too much of the work on UML semantics looks like a technical answer which is glad to have found a good practical question. We have asked what that question actually is, and refined it in the form of criteria for an improved definition. It is our hope that future work will explicitly address the larger task of improving the definition of UML.
Chapter 8

How UML Should be Defined

In Chapter 1 we set ourselves the task of solving the UML “definition problem”. That is, we wanted to find and justify a way of defining UML clearly, precisely and understandably. The productivity of development workers depends critically on this, because at present, much time is spent clarifying UML instead of analysing the subject matter of the required system. Agreements are both difficult to reach, and once reached, fragile because they are based on divergent understandings of a model. One of the main conclusions we drew from our detailed investigations in Chapter 5 is that the current definition [Obj07c] does define the language, including its semantics, at least enough to determine that a simple apparently inconsistent model actually is inconsistent. However, the effort that it took to determine the meaning of that model was completely unreasonable. Thus, there really is something there to define, and it needs to be defined more clearly. Our definition problem is a real problem that is likely to have a solution.

We sought the most basic principles of the language as currently defined, and found a very close match with graph transformation systems. Applying graph transformation to UML semantics is certainly not a new idea, but previous work has not taken into account the detailed ideas present in UML 2.x. Our bootstrapping process in Chapter 3 avoided the circularity of the official definition, yielding a precisely defined metamodelling language for UML’s abstract syntax. A clear but faithful account of attribute and association end navigation was added in Chapter 4 solving the problem of unwanted symmetry in naive accounts. Although the purpose of Chapter 5 was to test the existing definition, our study revealed many ideas that we have not seen considered in the formalisation literature surveyed in Chapter 7. Chapter 5 also uncovered many points requiring clarification, and an aspect of UML that we probably ought to consider broken: the descriptive dynamics or “emergent behaviour” such as Interactions. Chapter 6 presented earlier work using dynamic logic which we found suggests a better treatment of this aspect.

In this final Chapter, we draw together our findings and point the way forward to a UML with an unproblematic definition. The first section gathers together the conclusions of this dissertation, outlining a rigorous formulation of UML as a
graph transformation system. This defines the syntax and static semantics using a
category theoretic commuting diagram, making use of the metamodel and the inter-
pretation functors we have introduced throughout. We also outline the prescriptive
dynamic semantics (Section 5.3). As we saw toward the end of that section, the
descriptive dynamics as currently defined are not satisfactory. The second section
summarises our suggestions for further research. In particular, it investigates us-
ing a logic for descriptive dynamics, as a foundation or replacement for OCL, and
as a reference point for abstraction and approximation. The second last section
summarises our findings, and the last concludes.

8.1 What a UML Model Means

This section recalls our conclusions about UML and how it ought to be defined.
It is a terse but systematic account of the main ideas. Rather than repeat our jus-
tifications and explanations here, we give references to them. The subsection on
the semantic domain contains a new idea: the metamodel and the semantic domain
model are merged to obtain an appropriate type graph for the graph transformation
system.

The argument of Section 3.1 supports the hypothesis that UML models and
system states are graphs. We found further support for this hypothesis and nothing
to refute it, so it stands.

Concrete Syntax and Parsing

The diagrams, or concrete syntax of UML is as officially defined. There may be
benefits to formally defining it [BH02], but this is a problem beyond our scope.
Parsing of the concrete syntax into model element objects is also as described in
the official documents.

Object diagrams can be read quite directly as graphs. Binary link icons are
ambiguous (Section 3.2). They always represent one link and two edges, one for
each end and labelled with the text shown there. However each of the two edges
could originate at the link or the object whose icon is shown at the opposite end of
the link node.

When several diagrams are used to express a model, each diagram shows part
of the graph. Some elements may appear in several diagrams. When an object
diagram is used alongside other diagram types to partly describe a model, it is not
read directly as described above. Rather, it is read as described in the definition,
with object icons representing InstanceSpecifications and so on. We showed in
Section 3.6 that this reading generalises the direct concrete one.

We therefore have a second definition of the abstract syntax of UML: those
graphs that can be obtained by reading a collection of valid UML diagrams. How-
ever, the abstract syntax is actually defined to be the instances of the UML meta-
model. It is not clear that the two definitions of the abstract syntax will coincide.
Figure 8.1: A UML model is a graph $m$ that both contains and instantiates the metamodel $MM$

There may be diagram collections that represent graphs that do not instantiate the metamodel, and there may be metamodel instances that are not expressible as diagrams. Completing the precise definition we propose would enable a definitive answer to these questions.

**Abstract Syntax and Instantiation**

A model consists of Classes, Properties, Associations, Generalizations, Interactions and so on. The metamodel contains a node for each of these kinds of model element. The model contains the metamodel, and each model element is connected to its metamodel element by a specially named instantiation edge (Section 3.5). The instantiation is correct when these edges are the node component of a graph homomorphism (Definition 5) from the model into its $\gamma$ interpretation (Sections 3.1 and 3.3).

UML allows an object to instantiate more than one class (Section 3.3). We thus construct the power-graph (Definition 7), and consider the set of outbound instantiation edges of a node to map it to the set of nodes at their other ends. To ensure proper inheritance, the homomorphism must be into the subgraph of the power-graph whose nodes are closed under the partial ordering defined by the models Generalizations. That is, each set of Classes must contain the Classes that generalise them. The original graph instantiates its metamodel if the resulting edge map extends to a label-preserving graph homomorphism into this sub-powergraph.

This is partly summarised by Figure 8.1. A UML model is a graph $m$ that satisfies this diagram, where $MM$ is the metamodel, $\gamma$ the interpretation functor of Definition 5 and $\xrightarrow{\text{lm}}$ is a graph homomorphism whose node component is determined by the instantiation edges in $m$. That is, the metamodel is in the model, which in turn instantiates the interpreted metamodel.

The metamodel, as a UML model is also required to contain itself (which it does) and properly instantiate itself. We indicated that this self-instantiation can work for a small metamodel fragment in Section 3.6.

**Property Valuation**

Each Property has a value for each of its *classifiers* instances (Chapter 4). For each node and each label, the outgoing edges are totally ordered, and the “raw” value of an object’s attribute $\text{attr}$ is the list of values at the other ends of edges labelled $\text{attr}$ (Section 4.2). The instances of a Classifier include the explicit instances of the
Figure 8.2: The type graph $tg$ is the smallest graph that includes the user model $m$ and maps onto the semantic domain model $SD$ via $t$, making the diagram commute.

Classifiers that specialise it. Evaluation of a Property for a given instance of its classifier must also take into account the Properties that subset it, and the forgetful functors specified by the Property attributes $isUnique$ and $isOrdered$.

UML includes actions such as ReadStructuralFeatureAction (Section 4.2), which read values from the system state. OCL also has terms that are evaluated in the context of a system state, and model transformation languages have similar expressions. These are to be defined in terms of the Property values just outlined.

**Semantic Domain**

The elements that may appear in a run-time UML system state should be specified by a run-time semantic domain model, like the fragments given “non-normatively” in the definition [Obj07c, §13.1]. Doing so defines the broadest class of system states, and is to be refined by a particular user model. So, where the semantic domain model may have a class Object, the refined graph would replace this by the classes from the model. Instead of system states that contain objects, the refined graph would allow only system states with those certain kinds of objects. Similar refinement would apply to all kinds of user specified run-time entities such as Events, Behaviors, Associations and DataTypes.

A category theoretic construction that achieves this is indicated in Figure 8.2. The diagram will be replaced in a moment by a more comprehensive one, but it serves to introduce the main ideas. The model $(P \circ \gamma)(m)$ is obtained, in dealgebraised, power-graph form, along with a homomorphism $s$ shown on the left, into the semantic domain model $SD$. This homomorphism constrains the interpretation of the model in the obvious way. For example each Class in the user model maps to Object in the semantic domain model. We then find the smallest graph $tg$ such that the diagram commutes and the right hand arrow $t$ is onto (epic).

Given a model $m$ and an adequate interpretation $s$, we claim that $tg$ exists and is unique up to isomorphism. We therefore use this diagram to define a function $\tau$ from models to graphs, so that $\tau(m) = tg$. A system state of a given UML model $m$ can then be defined to be a graph $ss$ satisfying the diagram in Figure 8.3. The analogy with the model’s instantiation of the metamodel in Figure 8.1 should be clear. The difference is that our interpretation of the model incorporates the semantic domain model.
Figure 8.3: A system state of a UML model is a graph that both contains and instantiates the model (where the interpretation function $\tau(m)$ is temporarily defined by $tg$ in the diagram of Figure 8.2).

Figure 8.4: Abstract syntax and static semantics defined.

Combining the diagrams of Figures 8.1, 8.2, and 8.3, we obtain a diagram which completely defines the abstract syntax and static semantics of the language. This diagram is shown in Figure 8.4.

On the left hand side we have the traditional $M_2M_1M_0$ hierarchy, with the higher levels included in the lower. The $M_3$ instantiation is the composite of the inclusion $MM \hookrightarrow m$ and $i_m$ (the top-most “L” in the diagram). Thus the metamodel is the decoded form of the metamodel. The horizontal arrows are the explicit instantiations given by `instanceOf` edges in the graph, $i_m$ being the model’s instantiation of the metamodel, $i_{ss}$ the system state’s instantiation of the model.

The morphism $S$ maps metamodel elements to the semantic domain elements that their instances represent. It is only partial, because many model elements such as Actions do not have run-time counterparts, and others like Properties are nodes in the model but edges in the system state. Nevertheless, we expect that this partial map is enough to ensure that the nodes that instantiate model elements are the kinds of run time elements they are intended to be. This seems a natural way to define the semantic mapping from metamodel elements to semantic domain elements.

We conjecture that the graph $\tau(m)$ is determined up to isomorphism by its context in this diagram. It contains the dealgebraised model and maps onto the se-
mantic domain model $SD$. It is thus an augmentation of the included dealgebrised model, adding run-time notions such as event occurrences (Section 5.3), enabled actions (Section 5.6) and so on.

Note that $MM$, $SD$, $S$ and $P \circ \gamma$ could be replaced with other graphs, partial morphisms and interpretation functions, to yield a definition of a different language. A very practical possibility would be to define a new abstract syntax, but map it into the same semantic domain as UML, yielding a domain specific language compatible with UML. Another possibility is to augment the semantic domain model to include concepts needed for a new application, then apply this diagram to define UML over the new semantic domain.

**Prescriptive Dynamics**

How a UML system can evolve is defined by a transition relation over the class of system state graphs defined above. Graph transformation rules should be given for each kind of evolution step. One for each of the 36 concrete actions [Obj07c, §11] such as CreateObjectAction and ReadStructuralFeatureAction, plus rules for the various kinds of causation such as procedure invocation and signal acceptance (Section 5.3). Thus what can happen is determined by the objects, their enabled actions and their event pools. Since UML is intended to describe distributed systems, these rules ought to be applicable in parallel, where appropriate.

**Descriptive Dynamics**

One does not usually think of object diagrams as dynamic, but their intended meaning in the context of a larger model often has a dynamic element. Object diagrams are used for different purposes, sometimes just as an example, sometimes as an initial state, sometimes as the desired outcome of a process. They could be used to express what must always eventually happen, what must never happen, what must be possible and so on. To express these modes formally, enabling automated semantic analysis, it is necessary to employ modal or temporal operators. We develop this idea in the next section.

The model elements represented by an object diagram are converted into a token graph by a functor similar to the one which decodes the class model. The object diagram is satisfied by a system state when a label and Class preserving homomorphism exists from the token graph into the system state (Section 5.6).

Interactions, as they are currently officially defined, work similarly to object diagrams. A graph can be extracted from the model which will have a homomorphic image in the system state when the events it describes have happened (Section 5.4). The exception, as we pointed out in that Section, is that some Interactions involving deletion events are unsatisfiable even though the sequence of events they describe can happen. Like object diagrams, there are different ways in which an Interaction can be intended to constrain the model’s meaning, and these variations require modal or temporal operators to be made precise. Furthermore, an alterna-
tive, bug-free definition of Interaction semantics could be given using a temporal language over the UML transition system. The next section will outline this.

8.2 Towards a Well Defined and Understood UML

We set out to find a way to make the meaning of UML precise, a way that is faithful to the existing language, supports clear definitions of development process ideas such as refinement, is flexible enough to cope with profiles and semantic variation points, and above all, improves people’s understanding of models. We have studied each of these requirements and the language itself and proposed the outline of a definition.

This we hope is a step towards a world where analysts and their customers can conceptualise, explore and solve their problems without losing time and energy grappling with the tangle of ideas embodied in their modelling language. In this section, we describe the remaining steps needed to reach such a utopia.

Completing the Definition

The first step is to actually complete the definition we have outlined. Due to the size of the language and the need to support tools, this would need to be a software exercise rather than a paper one. Dependence on theory and technology beyond the underlying mathematics should be minimised.

In Section 4.2 we introduced a new kind of graph to accommodate the collection structures of UML’s static semantics. In order to define UML using these list-graphs, we would require a theory of list-graph transformation systems. We have not defined this, and we have not formulated a suitable category of list-graphs in order to apply the general graph transformation theory of [EPT06].

To facilitate independence from specific technologies, a good starting point would be to define a language for specifying graph transformation systems. This would allow independent implementations to interchange specifications. Work has been done on this [Tae01, Lam05], and the existing XML based language GTXL might be suitable. A model transformation language such as Kermeta [IRI], or lower level tool such as XSLT [Cla99] could possibly be used to obtain the graph transformation system definition from the metamodel and semantic domain model.

A modest but practical initial goal would be to mechanically establish the consistency of a corrected version of the example from Chapters 5 and 6 using an implementation of a partial definition. A simple way of doing this would be to implement the graph transformation system in the style of the HOL interactive theorem prover [Har]. This has a datatype for theorems and each of the logical rules are implemented as a function. The language ML [MTH90] used to implement HOL, was invented for this kind of syntactic rule system, and code can be kept very close to traditional mathematical definitions. We have found using ML in this way helps when developing formal systems because it provides interaction and it-
eration to the process. Model consistency could be demonstrated by constructing an initial state, confirming that it is correctly instantiated, then applying the steps represented by the Interaction. We have undertaken work along these lines, but not with great faithfulness to the existing definition.

A sequence of increasingly substantial example models could be tackled next, incorporating more and more of UML’s features and exploring possible model executions until we are confident that the definition works and agrees with the informal official one.

ML programs are very close to HOL formulae, so the definition implementation could be used to enable deductive verification. Perhaps it would be even better to code the definition in HOL, then generate the ML code from there using the work of Berghofer and Nipkow [BN02]. In this way, we could prove properties of the system in general and be very confident that the results apply to the generated implementation.

A Logic for UML

Discussions of logic are scattered throughout this study. Sections 1.1 and 1.3 gave a broad introduction to logic and its role in describing scientific models, and Section 6.1 introduced technical details of some actual logical systems. In Sections 5.3 and 5.4 we found grave difficulties with Interactions as currently defined, and mentioned the idea of replacing that treatment with a temporal or dynamic logic. Chapter 6 formalised our example model using dynamic logic, and the conclusion of that chapter, Section 6.4, explored the proposal of logic for Interactions in more detail. Then Section 7.3 in our survey chapter reviewed several logic related contributions to the UML formalisation literature.

In Section 7.3 we identified three main roles for logic in UML: constraints, Interactions and tool support.

We have not given much consideration to UML’s existing constraint language OCL, the logic-like language associated with UML. What status should we give OCL? We could treat its existing informal definition as we have treated that of UML, as something to be studied and faithfully made precise. Indeed, a formal semantics appears, with “informative” status in the official definition [Obj06b, Annex A]. Perhaps we should have used this as the starting point for the integrated semantics?

However, there are other requirements for a logical language over the UML semantic domain. A modal or temporal logic could be used to repair the broken Interaction semantics, as discussed in Section 5.4 and Chapter 6. We suggested in Section 6.4 that UML’s dynamics would be defined by a graph transformation system, generated by a set of rules, one for each concrete action and one for each kind of causation. The transitions in this system would be labelled by the names of these rules, along with the key participants in this particular application of the rule.

Thanks to Michael Norrish for this idea.
Thus, these action/causation labels could be the atomic programs for a dynamic logic, which would allow us to express Interactions in the style of Chapter 6, except over a faithful precise rendering of UML’s semantic domain.

The logic could have graphs as atomic propositions, which would be satisfied by a system state iff they map homomorphically into it. Thus, the specific model being considered could be asserted as a graphical axiom. These graphs would be expressed as object diagrams for general system state expressions, or as any collection of UML diagrams for models. For example, we could then use the obvious pair of object diagrams to state that any employee who has a desk also has a chair. A body of work on “graphical constraints” has studied the use of graphs as propositional atoms in temporal logics interpreted over graph transformation systems [HW95, GHK00].

A UML logic could also provide a basis for formal analysis techniques, especially model checking, automated and semi-automated theorem proving. A language sufficiently expressive to state OCL-like constraints is bound to be undecidable. Useful results could be obtained in such a language by interactive semi-automated theorem proving, but full automation will require abstraction or approximation. Fragments of logical languages often have less computationally complex decision problems. To restrict a language is to abstract its semantic domain, because certain features can not be expressed and are thus effectively ignored. Much of the study of logics is finding subsets of languages that can express useful statements, yet are computationally tractable. The sensible way to proceed is to begin with a language that can express everything likely to be needed, and then consider abstractions for specific purposes.

It will be important to find a syntax acceptable to developers. This is largely independent of the conceptual structure of the language. For example, the principle authors of OCL responded to frequent complaints about its awkward syntax by producing an alternate syntax similar to that of SQL [WK03, Appendix C] (We discussed SQL in Section 7.1). If we start with a “clean slate” and develop an expressive language to fulfill these requirements, then much of OCL could probably be recovered as an alternate syntax for this language.

System development often begins with use cases, an idea incorporated into UML. Development of a logic would also benefit from this approach. Below is a small selection of example formulae to guide us in identifying the required language features.

1. multiplicity constraints, eg. \((\forall \text{dog : Dog}) | \text{dog.owner} | = 1\)

2. Interactions, see Section 6.2 (on Page 123). Also, the mode of assertion needs to be expressed: this can happen, this can always happen, this will always happen given this initial state, . . .

3. invariants, often relating associations, eg. \((\forall x : X) x.a.b \subseteq x.c\)

4. safety property \(\text{Init} \rightarrow □¬ \text{Bad}\)
5. liveness property, e.g. $\text{AF}(\text{chocolate} \lor \text{refund})$ (The CTL path quantifier AF means every computation eventually reaches a state satisfying the argument)

We see that the logic will probably need variables and quantification. What should these variables range over? In [BCG04], the variables range over hyper-edges. In our formulation, we would need quantification over nodes, as we saw in Section 6.4. Would we need to quantify over edges, or collections?

We have not discussed types in depth, but merely followed UML’s typing ideas, and the notion of a type-graph from graph transformation theory. The theory of object oriented typing systems is studied in [Bru02], and this work has been related to the UML setting in [SJO7]. Would there be value in a more elaborate typing system in the proposed logic?

Since UML allows concurrency, and this is supported by graph transformation, a single transition could be labelled with many concurrent actions, eg. $S \xrightarrow{a,b} S'$. When evaluating the formula $\langle a \rangle P$ in state $S$, would $S' \models P$ make this formula true? Many details like this would need working out.

The interaction of modalities and quantification is notoriously complicated [Fit99]. Consider the apparently simple sentence “everything is always good”. When we say “everything”, do we mean everything that exists now, or do we also require that things that come into existence later are also good? When something ceases to exist does it also cease to be “good”? Even predicates, such as “good” can be allowed to have different meanings at different points in time. Thus there is another choice in interpreting our example sentence, do we mean “good” in the present sense, or does the meaning vary according to the states being considered. Fitting proposes a quantified modal logic which allows each of these choices to be made explicit in the formula [Fit99]. He applies this to several well known philosophical problems, such as Frege’s puzzle about how “the morning star is the evening star” can be informative even though it is a tautology, and Russell’s attempt to find a truth value for “the king of France is bald”, even though there is no king of France. The domain of quantification in a UML system is varying, neither strictly increasing nor decreasing, which makes it a complicated case in terms of quantified modal logic. However, we see no need for predicate and functions symbols to vary over time.

The key non-logical operation we see in the examples above is navigation of Properties, which we defined in Section 4.2. Generally, the result of evaluating a property for a given object is a collection, a list, bag, ordered set or set. Integrating the theory of collections and Property navigation will be an important part of defining the logic.

As we have pointed out, the full logic is bound to be undecidable, and so fragments of the language, or abstractions over the semantic domain will be required for automated tool support. Contributions proposing automated solutions to UML verification problems routinely code up a partial approximation of some aspect of UML into a formal language with tool support. What this amounts to is describing
a model of the model, but there is no systematic account of how the result resembles the original. What questions can be reliably answered using the formalisation, and what information is lost or distorted? This suggests a theoretical question that goes well beyond UML, and to the very essence of modelling. In any case, understanding the preservation properties of the required abstractions would be much more feasible if the original semantic domain was precisely defined, and a highly expressive reference language defined over it.

8.3 Findings

This section briefly summarises our findings.

**Official Definition**

Our detailed study and use of the official definition revealed the following.

1. the official definition for UML 2.x [Obj07c] describes the semantics of the language in enough detail to determine that some models are inconsistent (Chapter 5, especially Section 5.7)

2. the definition is very difficult to understand (Section 5.7)

3. many specific points require clarification (Section 5.7), and at least one aspect of the language is incoherent (Section 5.4, the event-rot bug, see below)

4. the semantic domain described in the official definition closely resembles a graph transformation system (Sections 3.1, 5.6)

**Definition Criteria**

In Chapter 2 we argued for criteria for an improved definition of UML. They are briefly summarised below

0. **faithful** to existing UML definition

1. **understandable** and implementable (supports agreement and tools)

2. **defines** concrete syntax → abstract syntax → semantic domain

3. **clarifies** ideas: model consistency, model refinement, transformation soundness

4. **reflective** - objects can access their classes

5. **flexible** to support profiles and variations
Bootstrapping

1. the official definition is circular, using UML to define UML
2. we employ graphs in a bootstrapping process which recovers the intent of the official definition without circularity (Chapter 3, especially Sections 3.2 and 3.6)

Dealgebraisation and Interpretation Functors

In Section 3.3, we introduce the dealgebraisation functor into the definition of instantiation. This has the following benefits.

1. recovers the intuition of an object diagram instantiating a class diagram, which is lost when we recognise the form of the class diagram fragment of the model
2. enables a non-trivial account of self-instantiation, some models instantiate themselves, but others do not
3. enables two or more levels of instantiation within the one graph
4. non-trivial self instantiation and multiple levels of instantiation are also achieved in [GFB05], but the repository form of the model is lost, eg Properties are edges, not nodes
5. we suggest that other interpretation functors could select and transform the relevant model elements, giving a semantics by diagram type, the idea is explored for object diagrams and sequence diagrams (Section 5.4)

Collections, Property Values and Association Symmetry

In Chapter 4 we develop a theory of Property values by building on the work of Dragan Milicev [Mil07].

1. we define a new kind of graph, the list-graph, motivated by the need to represent UML’s hierarchy of collection structures
2. many questions are left open on list-graphs, especially whether the graph transformation theory of [EEP10] can be applied to them
3. we develop a more uniform version of Milicev’s Property semantics, dispensing with ad-hoc treatment of ordered Properties
4. our account removes unwanted symmetry from association end annotations \{union\} and \{subsets x\}
Semantic Domain Model and Type Graph Construction

Many authors define the space of system-states by using the model as a type-graph, but generally do not take into account the details of UML 2.x as officially defined.

1. we handle generalisation by composing a power-graph functor with the dealgebraisation functor mentioned above (Section 3.3)
2. a subgraph of the power-graph is described to ensure correct instantiation of super classes, but this should be formulated more carefully and checked with some examples
3. we propose a formal and “normative” role for (a possibly revised and enlarged version of) the non-normative semantic domain model of [Obj07c, §13.1] (Section 5.3)
4. we give a construction of a type-graph from the model, the metamodel and the semantic domain model (Section 8.1), examples and theoretical study are needed to confirm that the construction works as intended

Descriptive Dynamics and the Event-Rot Bug

1. in Section 5.4 we discovered a bug in the semantics of Interactions: recording the history of the system in the system state fails when objects are deleted
2. it may be possible to correct this by distinguishing between deleted and “existing” objects
3. however, we suggested replacing the “recorded history” approach by a dynamic logic

Dynamic Logic

We found several reasons to consider using a dynamic logic as part of UML’s definition, including the event-rot bug mentioned above. We began to investigate this possibility (Section 6.4).

1. modalities are needed for practical uses of object diagrams, eg. this situation will occur, always occurs, must not occur (Section 8.1)
2. OCL should be either properly defined or replaced
3. a definitive logic for UML would take the guesswork out of formalisations for logic based tool support such as model checking and deductive verification, much work remains to be done on this (Section 8.2)
8.4 Conclusion

Section 8.1 outlined our solution to the UML definition problem, a solution justified by our sustained careful analysis of the language as currently defined. The problem was introduced in Chapter 1 where we noted that the lack of clarity in the current definition is a serious drain on development productivity. Chapter 5 provided evidence that the definition is indeed very unclear, but also that there is meaning to be found there, to be expressed more clearly.

The test performed in Chapter 5 was to find the meaning given by the official definition [Obj07c] to a tiny example model that appeared to be inconsistent. Preparations required for this test were extensive. In Chapter 2 we investigated the nature of modelling and UML style modelling languages, and determined what would be required from a good language definition. The result was our 6 Criteria, summarised in Table 7.1 (on Page 131). In order to talk sensibly about the model, we needed to find semantics for the fragment of UML used to define the abstract syntax of UML. Chapter 3 looked to the definition for clues about the semantic domain, and found that the core of the UML metamodel is the category used to define edge-labelled graphs, and the usual intuitive interpretation of it mirrors that mathematical definition. We were able to redevelop the ideas in the official definition in a way that avoids its apparent circularity. The result was a graph theoretic rendering of a small static fragment of UML, including its semantics. That left a few tricky details to be sorted out in Chapter 4, in particular, an account of Property values that avoided unwanted symmetry of association end annotations, and a proper account of generalisation.

The point of Chapter 5 was not only to see whether the apparent inconsistency of the example model was realised in the definition, but to guide us in a purposeful way through the definition, to develop a thorough understanding of the central ideas of UML. We are of course aware that UML was written by a committee, and has evolved under an evolving committee, and can be expected to be a little incoherent conceptually. Nevertheless, core ideas did emerge, and fitted nicely with our discovery early in Chapter 3 that the semantic domain described in the definition looks a lot like a graph. Even the descriptive dynamics of the language seemed to be rendered as graphs even though this turns out not to make good sense. Maintaining a recorded history of the system in each system state as event occurrences joined to their participants has the fatal flaw that the record deteriorates as the participants do. Either the record must be separated from the actual participants, or the descriptive dynamics should be expressed using a form of temporal or dynamic logic.

Our literature survey appears near the end of the dissertation, in Chapter 7, because we considered it necessary to understand what a good definition is (Chapter 2), and what it is that needs to be defined (Chapter 5) before assessing candidate definitions. Chapter 6 is effectively part of this survey, because although it is our own work, it comes from the period before the main ideas of the thesis. It sought to use dynamic logic to define UML, by translating sets of diagrams into dynamic
logic formula. This enabled a tableau style search to test the model for consistency (satisfiability). However, our present interest in this work is its treatment of Interactions as dynamic logic formulae. The conclusion of that chapter outlines an alternative to the official approach based on this idea. In the survey chapter, database theory was presented as an example for UML to aspire to. Its clear and simple conceptual foundations have had clear technical benefits, but also aids its understandability. Previous graph transformation work was surveyed, and we found much good in the dynamic metamodelling approach [EHHS00]. The technical aspects of our own work could be seen as a development of this theme, aligning it better with the ideas of the current official definition. Various other contributions were considered, using various other techniques. Two of these works [Obe03] and [MB06, BCR07] stood out as faithful and plausible attempts to better define UML. We spent some time contemplating the lack of elegance noted by the authors of the second work, and suggested that what was needed was a simple mathematical idea that captures the essence of the language. We believe that graphs and graph transformation systems are the required structure.

In this concluding chapter, we summarised our findings, merging several related partial definitions into a category theoretic commuting diagram in Figure 8.4 which gives a modular and thus flexible definition of the abstract syntax and static semantics of UML. Further required work was outlined, in particular the development of a logic with the UML graph transformation system as its semantic domain. This logic would serve as a constraint language, to sensibly formulate descriptive dynamics such as Interactions, and as a reference point for the abstractions and approximations needed to provide automatic tool support for verification and validation tasks.

We set out to find and justify a way to clearly, precisely and understandably define UML. As we saw in Chapter 7, almost every conceivable formal apparatus has been applied to UML, but we looked for the answer in the current definition, rather than the handbook for our favourite formal technology. We have found an answer in graph transformation systems, which were unknown to us when we began this search. Unsurprisingly, researchers have already investigated the method we propose, but what we have achieved is a strong justification of this choice. Proposals for different solutions must now show that they better meet our criteria from Chapter 2 or challenge those criteria. Any claim that another formal system is more faithful to UML as it is now defined must challenge our detailed analysis in Chapters 3 to 5. Debate on these topics would advance the state of research, by focusing it more clearly on the problem to be solved. However we hope that it will not be necessary. We hope that we have convinced our readers, that UML wants to be a graph transformation system.
Bibliography


[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of java card programs. In Java on Smart Cards: Programming and Security, number 2041 in LNCS, pages 6–24. Springer, 2001.


