PhD Thesis

The Design of Formal Languages

Ian Salter

University College London
Abstract

The aim of the thesis is to develop a framework to support the design of formal languages. The thesis consists of two parts. The first part attempts to develop a conception of formal language design. The conception considers the nature of formal languages and acts as a specification for a framework to support the design of formal languages. The second part develops the formal aspects of such a framework.

The first part considers the nature of formality and the nature of disciplines. Formality is considered in terms of different philosophies of mathematics, it illustrates how these different philosophies give rise to different notions of formality, and leads finally to a strongly relativistic definition of formal language. The nature of disciplines is considered in terms of philosophies of science resulting in the definition of a generic engineering conception of design disciplines. The definition of formal language is used to instantiate the generic engineering conception resulting in the conception of formal language design.

The basis of the framework that forms the second part of the thesis is the Z notation enriched with Category theory. This notation is used to instantiate the conception outlined in the first part of the thesis. Pre-order categories are advocated as the basis for representing conflicting requirements for formal languages. Category theory is used to develop a generalised notion for defining the syntax of languages that, when used by appropriate agents, satisfy language requirements. According to the conception, knowledge to support design is embodied in engineering principles. Categorial notions are used to describe the formal and empirical components of engineering principles.
# Contents

Abstract ......................................................................................................................... 2  
Contents ........................................................................................................................ 3  
List of Figures .............................................................................................................. 6  
Acknowledgments ...................................................................................................... 9  

1 Introduction .............................................................................................................. 10  
  1.1 Motivation for the Thesis ......................................................................... 11  
  1.2 The Class-Instance Distinction .......................................................... 14  
  1.3 Dimensions of Epistemology ............................................................. 18  
  1.4 Overview of the PhD .......................................................................... 22  
  1.5 The Conception of Formal Language Engineering ............................ 23  
  1.6 The Framework for Formal Language Design .................................... 24  
  1.7 Criteria For Assessment .................................................................... 26  
  1.8 Conclusion .......................................................................................... 27  

2 On the Nature of Formal Languages ..................................................................... 28  
  2.1 Philosophy of Mathematics .................................................................. 29  
  2.2 The Notion of Formal Language .......................................................... 37  
  2.3 Conclusion ............................................................................................ 43  

3 On the Nature of Frameworks ............................................................................... 44  
  3.1 Philosophy of Science ........................................................................... 45  
  3.2 Science and Design ............................................................................... 54  
  3.3 A Generic Engineering Conception ...................................................... 61  
  3.4 Formality and Design ............................................................................ 70  
  3.5 Conclusion ............................................................................................. 78  

4 An Engineering Conception of Formal Language Design ................................. 79  
  4.1 Outline of the Scope of the General Design Problem ......................... 80  
  4.2 Requirements ......................................................................................... 83  
  4.3 Artifact ................................................................................................... 85  
  4.4 Related Disciplines ............................................................................... 87  
  4.5 Language Engineering and HCI Engineering ....................................... 89  
  4.6 Conclusion ............................................................................................. 92  

5 A Basis for the Formal Framework ........................................................................ 94  
  5.1 Choice of a Formal Framework ............................................................ 96  
  5.2 The Z Notation ....................................................................................... 101  
  5.3 Graphs, Categories and Diagrams .......................................................... 110  
  5.4 Monics, Epics and Isos .......................................................................... 126  
  5.5 Functors and Categories of Categories ............................................... 132  
  5.6 Limits and Co-limits ............................................................................... 141  
  5.7 Conclusion ............................................................................................. 153
List of Figures

Figure 1.1 Extreme Rationalism .............................................................. 18
Figure 1.2 Extreme Empericism ............................................................. 19
Figure 1.3 Extreme Absolutism .............................................................. 20
Figure 1.4 Extreme Relativism ............................................................... 21

Figure 3.1 The Naive Inductivist Framework .......................................... 45
Figure 3.2 The Naive Falsificationist Framework .................................... 47
Figure 3.3 The Lakatos Framework ......................................................... 49
Figure 3.4 The Kuhn Framework ............................................................ 51
Figure 3.5 The Scope of the General Problem for HCI (taken from [Long & Dowell 89]) ......................................................... 54
Figure 3.6 Conception of HCI as a Craft Discipline (following [Long & Dowell 89]) .......................................................... 56
Figure 3.7 Conception of HCI as an Applied Science Discipline (following [Long & Dowell 89]) ................................................ 57
Figure 3.8 Science, Applied Science and Design, the Relationship between Kuhn and Long & Dowell ........................................ 58
Figure 3.9 Conception of HCI as an Engineering Discipline (following [Long & Dowell 89]) ..................................................... 59
Figure 3.10 The Phenomena of Design Problems .................................... 63
Figure 3.11 Design Problems (Phenomena and Shared Exemplars) ......... 64
Figure 3.12 Design Problems (Practice and Research Exemplars) ......... 66
Figure 3.13 The Generic Conception of Engineering Design ................. 68
Figure 3.14 A Naive Conception of Program Verification ....................... 70
Figure 3.15 A Revised Conception of Program Verification .................... 71
Figure 3.16 Formal Verification of an Algorithm .................................... 72
Figure 3.17 Empirical validation of a program ....................................... 72
Figure 3.18 Executable Specification ..................................................... 73
Figure 3.19 The Relation between Specification and Prototype ............. 74
Figure 3.20 The Generic Engineering Conception .................................. 75

Figure 4.1 An Example Formal Language Worksystem .......................... 81
Figure 4.2 Requirements for a Formal Language Worksystem ............... 81
Figure 4.3 The Form of the Requirements Specification of Agent 1 of Figure 4.1 .......................................................... 86

Figure 5.1 The Scope of the Formal Framework ..................................... 94
Figure 5.2 Existing Formalisms and their Positions within the General-Specific Spectrum .......................................................... 97
Figure 5.3 An Ordering of Design Preferences ...................................... 110
Figure 5.4 A State Transition Diagram .................................................. 111
Figure 5.5 Figure 5.3 with Composite Relationships .............................. 112
Figure 5.6 Figure 5.4 with Composite Relationships .............................. 112
Figure 5.7 An Operation Demonstrating Abstraction ............................ 133
Figure 5.8 An Example of a Pre-order Category ................................. 142
Figure 5.9 All the Elements that are Less or Equal to Both a and b ......... 142
Figure 5.10 In Figure 5.8, d and e are Less Desired than c ..................... 145
Figure 6.1 PSBenefit .....................................................................................................163
Figure 7.1 The Attributes of the Real Interval..........................................................181
Figure 7.2 The Attributes of the Cuboid .................................................................182
Figure 7.3 The Attributes of the Cylinder .................................................................184
Figure 7.4 The Dependencies Between Components in the Specification of ATCWorkStructures ........................................................................188
Figure 7.5 The Attributes of a Beacon .......................................................................189
Figure 7.6 The Relationship Between Fuel Consumption Rate, Speed and Cruising Speed ..............................................................................191
Figure 7.7 The Sector, Beacon and Corridor Attributes of an ATC Sector ......................................................................................................195
Figure 7.8 The Legal Sector Space for the ATC Sector of Figure 7.7 .....................196
Figure 7.9 An Example of a Position Attribute of a Basic Actual Flight ..................199
Figure 7.10 Examples of a Speed and Fuel Consumption Attributes of a Basic Actual Flight ................................................................................200
Figure 7.11 An Example of a Separation Space Attribute of a Basic Actual Flight over a Time Period .......................................................200
Figure 7.12 An Example of the Possible Predictions of an Actual Flight ............203

Figure 9.1 The Text-Based Structure Class as an Instance of the schema StructureClass .......................................................................................283
Figure 9.2 The Shape Structure Class as an Instance of the schema StructureClass .......................................................................................285
Figure 9.3 The Graph Structure Class as an Instance of the schema StructureClass .......................................................................................287
Figure 9.4 The Picture Structure Class as an Instance of the schema StructureClass .......................................................................................289
Figure 9.5 A Graphical Representation of the Production Rule B -> DB ............309
Figure 9.6 The Graphical Representation of the Derivation of 1DB from 1B using Production Rule B -> DB.......................................................310
Figure 9.7 The Simplified Graphical Representation of the Derivation of 1DB from 1B using Production Rule B -> DB .......................................................311
Figure 9.8 A Demonstration that the String 01 is a part of the Language Defined by the Text-based grammar of Sub-section 9.1.1 .......................312
Figure 9.9 The Simplified Graphical Representation of a Shape Grammar Derivation ............................................................................314
Figure 9.10 The Graphical Representation of a Graph Grammar Production Rule ....................................................................................320
Figure 9.11 The Graphical Representation of a Graph Grammar Derivation ..........321
Figure 9.12 The Simplified Graphical Representation of a Graph Grammar Derivation ....................................................................................322
Figure 9.13 The Simplified Graphical Representation of a Picture Layout Grammar Derivation ....................................................................................323
Figure 9.14 The Generation of a Venn Diagram .......................................................331
Figure 11.1 The Formal Aspects of the Generic Engineering Framework .......................................................... 351
Figure 11.2 The Empirical Aspects of the Generic Engineering Framework .......................................................... 354
Figure 11.3 A Fragment of the Category of Requirements Specifications .......................................................... 365
Figure 11.4 A Fragment of the Category of Artifact Specifications .......................................................... 366
Figure 11.5 A Fragment of a Framework Category ....................................................................................... 368
Figure 11.6 Defining the Formal Components of Principles extends the Composition Operator in the Framework Category .......................................................... 369
Figure 11.7 A Basis for the Formal Application of a Principle .......................................................... 370
Figure 11.8 The Formal Application of a Principle ....................................................................................... 371
Figure 11.9 The Formal Aspects of Principle Operationalisation .......................................................... 371
Figure 11.10 A Basis for Generalising Successful Craft-based Design .......................................................... 373
Figure 11.11 Generalising Successful Craft-based Design .......................................................... 374
Figure 11.12 A Basis for Combining Principles ....................................................................................... 376
Figure 11.13 Combining Principles ....................................................................................... 376
Figure 11.14 The Basis for Assessing Successful Principle Application .......................................................... 380
Figure 11.15 Successful principle application. ....................................................................................... 380
Acknowledgments

Thanks are due to my academic supervisor John Long, for his faith, patience and encouragement. I hope that the submission of the thesis will not mark an end to the discussions that typified supervision meetings. I must also thank Allan MacLean, my industrial supervisor from Xerox EuroPARC, for his help and encouragement. I hope the work justifies his time and Xerox's expense.

Thanks are also due to John Dowell for the collaboration on the Air Traffic Control work, as well as discussions concerning the HCIE conception. For similar discussions thanks must also go to Becky Hill, Wally Smith and other members of the Ergonomics Unit at University College London between 1990 and 1994.

Finally, thanks are due to the rest of my family. To my father Ray, for his help with proof reading. To my mother Irene, for her gentle encouragements and assistance with the proof reading. To my children Jessica, Maxime and Tegan for distraction and inspiration. Most of all, to my partner Rachel, whose love, sacrifice and support have made the whole enterprise possible.
1 Introduction

This chapter introduces the aims and objectives of the thesis. It considers some of the key concepts that are used as part of the thesis. It outlines the different components of the thesis. Finally it establishes the criteria by which the thesis is assessed in the conclusion.

Briefly stated the aim of the thesis is:

To develop a framework to support the design of formal languages.

This statement could be interpreted in many ways. The term formal language is open to different interpretations (see Chapter 2). There are different conceptions as to what it means to support design (see Chapter 3). The first three chapters of the thesis are all an attempt to elaborate upon this stated aim.

This chapter contains the following sections:

The Class Instance Distinction
Dimensions of Epistemology
Overview of the Thesis
The Conception of Formal Language Engineering
The Framework for Formal Language Design
Criteria For Assessment
Conclusion

The first section considers the personal motivation behind the thesis. It illustrates how the initial aims have evolved during the three years that it took to write. It is intended that it will give the reader an early insight into the nature of the work. On account of its personal nature, this section is written in the first person.

The second section introduces the class-instance distinction that is used throughout the thesis as both a macro level structuring mechanism and a micro level tool of analysis. The third section introduces two dimensions of epistemology that lie behind much of the work of the early chapters.
The fourth section presents an overview of the thesis exploiting the class-instance distinction introduced earlier to explain the different parts. The fifth section outlines the first part of the thesis which introduces a conception for formal language design. The conception acts as a specification for the framework that is developed in the later chapters of the thesis. The sixth section outlines this framework.

The final section establishes the different criteria by which the PhD is assessed in the conclusion.

### 1.1 Motivation for the Thesis

At the beginning of the period of study that has led to this thesis I had just completed a period as a lecturer in computer science at a London polytechnic. My undergraduate training was primarily in mathematics. Thus, in many senses I began the study for the PhD strongly schooled in a rationalist tradition. This tradition is perhaps best exemplified by the following quote from one of the foremost advocates of formal methods in software engineering:

"Computer programming is an exact science, in that all the properties of the program and all the consequences of executing it can, in principle, be found out from the text of the program by purely deductive reasoning" [Hoare 69]

As a mathematician, however, probably in contrast to many advocates of formal methods, I was, essentially, what might be called a visual reasoner. I found little difficulty in proving the theorems of mathematical analysis since I could visualise the theory geometrically and construct the required proof accordingly. With algebra, where proofs revolve heavily around symbol manipulation, I was not so effective.

As a lecturer, I had developed a variety of non-textual tools to aid explanation in the courses for which I was responsible. In the first year programming courses, I introduced a simple graphical notation for expressing algorithms [Salter 89], which although not empirically tested, was well received by both students and staff. In the formal methods courses, students were encouraged to draw pictures of logical formulae to illustrate aspects, such as the binding
of variables, which, although well understood by mathematicians, is often problematic to computer scientists.

It seemed to me that work in the design of languages had concentrated largely upon, what at the time I would have called, the construction of a theory of text based languages. Thus, my aim at the start of the PhD was:

To develop a theory of graphical languages.

I used the term language to mean formal language. Such a theory, it seemed to me, would be useful in the development of notations for computer system design and notations for constructing mathematical entities. In addition it would provide another perspective on Human Computer Interaction (HCI) since, it is my view, all successful interaction that occurs between human and computer occurs in some formal language.

HCI was an active area of interest and, from what I knew then, not many were concerned with a graphical theory of language. A research group that was interested in HCI might provide an ideal environment in which to pursue the PhD. In addition, an HCI research group in the empirical tradition would also provide a novel perspective to the work.

I arrived at the Ergonomics Unit, University College London, largely by chance and still marvel at my good fortune. Not only did I become involved in a research group in the empirical tradition, but also one concerned with the meta issues of HCI design. The Unit was interested in the nature of the design practice of HCI and the nature of the knowledge to support that practice.

The first thing that came into question in this environment was the notion of a theory. To me, as a mathematician, a theory was a set of axioms expressed in some formal system, such that theorems could be derived from the theory. Theories were a formal expression of some knowledge. Much of the early work of the PhD was involved in exploring the nature of knowledge, and in particular what form of knowledge would best support design. From this work, I established a generic specification of an engineering framework to support design. This work is presented in Chapter 3. The framework is intended to be prescriptive rather than descriptive, although I firmly believe that no one should be forced to take the medicine.
The second thing that came into question was the notion of formality, and what was meant by a formal language. At the beginning of the thesis, something was formal to me if it was clear and unambiguous. A formal language was thus a language in which everything was clear and unambiguous. One might have asked how is it possible to determine if a language is formal? My response would have been to detail various mathematical means by which the properties of a language might be established. However, with the study of the nature of knowledge proceeding in parallel, I began to question such an entirely rationalist approach. As I considered empirical means for establishing the formality of a language, it became apparent to me that it was impossible to establish an absolute notion of formality. Thus, Chapter 2 defines the formality of a language relative to a class of agents, that interpret and employ that language relative to some purpose.

The questioning of the notion of formality had led to a brief survey of different definitions of formal language. Two surprising things emerged from this study. The first was the number of basic text books dealing with formal languages of one form or another that employed the term formal language without defining either formal language or even formality. The second was that when definitions of formal language were given, they, either explicitly or implicitly, seemed to consider only textual representations as formal. Even when the definitions could be interpreted as applying to non-textual representations the authors shied away from asserting that such representations could be formal. With the definition that I now advocate the formality of a language could be tested empirically and thus in my terms a formal language could be any form of representation. Thus, I dropped the word graphical from the aim of the thesis.

The result of these deliberations was to modify the aim of the thesis. It now aimed:

To develop a framework to support the design of formal languages.

The first half of the thesis presents a conception of the design of formal languages. The conception acts as a specification for a framework to support language design. The conception has rationalist and empirical aspects. At the
meta level, I had taken on board the belief of the necessity of an empirical component to any knowledge and practices that support design. However, by training I remained a rationalist, and although this thesis outlines the role of empirical work, it remains a work of rationalism. There are no empirical studies, just formal specifications. Thus, the second half, rather than presenting a complete example of a framework that satisfies the conception, presents only formal aspects of such a framework. Thus the aim for the second part of the thesis could be stated:

To develop a formal framework to support the design of formal languages.

When I began the period of study for the thesis, I adhered to a philosophy of mathematics known as constructivism. A constructivist believes that mathematical entities are mental constructions rather than discoveries (see Chapter 2). Nothing in the last three years has shaken that conviction. The intention of this disclosure is to make explicit the key assumption with which the work was carried out. Although the formalisms employed are not necessarily constructivist, there is a constructivist feel running throughout the thesis, not least in the form of the class-instance distinction discussed in the next section.

1.2 The Class-Instance Distinction

This section considers the class-instance distinction that is used throughout the thesis as both a macro level structuring mechanism and a micro level tool of analysis. The class-instance distinction was introduced by Aristotle, although he employed the terms universal and particular.

"Now of actual things some are universal, others particular (I call universal that which is by its nature predicated of a number of things, and particular that which is not; man, for instance, is a universal, Callias a particular)." [Aristotle, Logic]

A class is therefore a collection of instances of that class. The constructive mathematician, Martin-Löf, has demonstrated if mathematics is treated constructively then many relationships can be viewed in terms of the class-
instance distinction. The following table is a recreation and extension of a table presented in "Intuitionistic Type Theory" [Martin-Löf 84a]. The extension to the table is based upon the work presented in "Constructive Mathematics and Computer Programming" [Martin-Löf 84b] and involves the specification-program relationship.

<table>
<thead>
<tr>
<th>Class</th>
<th>Set</th>
<th>Type</th>
<th>Problem</th>
<th>Proposition</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>Element of the Set</td>
<td>Instance of the Type</td>
<td>Solution to the Problem</td>
<td>Proof of the Proposition</td>
<td>Program fulfilling the Specification</td>
</tr>
</tbody>
</table>

A set can be seen as a class with the elements of the sets considered as class instances and a type can be seen as a class with any instance of that type as an instance of that class. A problem can be seen as a class, the class of all possible solutions to that problem. Of course, for a problem to be a class, it has to be sufficiently well-defined so that given a hypothetical solution, it is possible to determine if it is indeed a solution to the problem.

The notion of a proposition as a class derives from the constructive view of mathematics. Consider the following proposition:

For any natural number \( n \), and any natural number \( m > n \) there exists a natural number \( i \) such that \( i + n > m \).

To a constructivist, a proof of this proposition is a construction that provides an \( i \) for any \( n \) and \( m \) obeying the above conditions. Thus the functions \( f \) and \( g \) defined as follows are both proofs of the above proposition:

\[
f(n,m) = m - n + 1 \quad \text{if} \quad m > n \\
g(n,m) = m - n + 2 \quad \text{if} \quad m > n
\]

Propositions can be seen as defining a class whose instances are the proof of the proposition. The final notion derives from the link between constructive proofs and functions. Computer programs can be viewed as functions, and thus as constructive proofs, of some proposition. The proposition is thus a specification for the program. Thus, specifications represent a class whose instances are all the programs that fulfil the specification.
For Martin-Löf and other constructive mathematicians there is an equality relationship between each of the class-instance relationships presented above. A broader view is taken here, each relationship is seen as being of the class-instance form. In other words, each relationship is an instance of a class-instance relationship. It is possible to list further instances of the class-instance relationship. These instances are presented in the table below:

<table>
<thead>
<tr>
<th>Class</th>
<th>Theory</th>
<th>Specification</th>
<th>Requirements Specification</th>
<th>Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>Phenomena</td>
<td>Implementation</td>
<td>Artifact Specification</td>
<td>Practice</td>
</tr>
</tbody>
</table>

A (scientific) Theory can be seen as a class, whose instances are the explained and predicted phenomena of the theory. By broadening the specification-program relationship beyond computer programming, a specification in general can be seen as a class whose instances are implementations of the specification. The class-instance distinction can be used to distinguish between different sorts of specification, a requirements specification, for example, can be seen as a class whose instances are artifact specifications. The relationship between knowledge and practice in design can be seen in terms of the distinction. Thus, design knowledge representing a class whose instances are design practices.

As has already been stated, the class-instance distinction has been used as a meta level structuring mechanism in the thesis. The first four chapters aim to establish a conception of the design of formal languages. This conception is in a sense a specification of a framework to support the design of formal languages. Thus, the conception is a class whose instances are the different possible frameworks that are consistent with the conception. The latter half of the thesis outlines a formal framework which is part of an instance of the class defined by the conception.

Within the formal framework, the general nature of aspects of language design are described. Thus, Chapter 6 outlines the general form of the performance requirements of a formal language worksystem. As a partial attempt at validating the general form of performance requirements, specific performance requirements are constructed to be consistent with the general
requirements. Thus, the general performance requirements represent the class of all performance requirements for formal language worksystems.

The class-instance approach is also used at the micro level as a tool of analysis in the early chapters of the thesis. It is employed in the next section to derive the dimensions of epistemology that are used in the discussion of the different theories of mathematical and scientific knowledge that occur in the next two chapters.
1.3 Dimensions of Epistemology

The class-instance distinction can be used as a tool of epistemology - the branch of philosophy concerned with the study of knowledge. General knowledge can be viewed as a class and particular knowledge as instances of that class.

This section outlines two dimensions of epistemology, the rationalist-empiricist dimension and the absolutist-relativist dimension. These dimensions are viewed in terms of the class-instance distinction.

Extreme rationalism can be defined as the belief that it is possible to obtain knowledge by means of reason alone. Thus, for the extreme rationalist, knowledge may be general or particular and relationships can be established between the components of knowledge by deductive reasoning. To the rationalist, knowledge is constructed from agreed general and particular knowledge by means of deduction. This situation is represented graphically in Figure 1.1.

![Figure 1.1 Extreme Rationalism](image)

The classical philosopher Plato, although he predates the class-instance distinction introduced by Aristotle, may be thought of as an extreme rationalist. Highly sceptical of the senses, he believed that the objects of
knowledge were eternal and unchangeable forms. Objects perceived by the senses are merely imperfect copies of these forms. In the terminology above the forms would correspond to agreed particular knowledge.

In contrast to extreme rationalism, extreme empiricism implies a belief that knowledge can be obtained only through experience. Thus, general knowledge is derived inductively from empirical phenomena. The situation can be represented graphically in figure 1.2.

![Diagram of Extreme Empiricism]

The philosopher Comte, who coined the term positivism, may be thought of as an extreme empiricist. According to Leahey:

'As an epistemology, positivism adopted a radical empiricism. Metaphysical speculation and explanations of nature in terms of unobservable entities were to be abandoned. Instead, human knowledge would confine itself to collecting and correlating facts to provide an accurate description of the world. Such was the method and proper philosophy of science according to Comte.' [Leahey 87].

Extreme rationalism and extreme empiricism represent opposite poles along a rationalist-empiricist dimension. Different epistemologies are situated along
the dimension according to the relative emphasis they place upon reasoning or the observation of phenomena as a means of deriving knowledge.

The rationalist-empiricist dimension of epistemology is concerned with the means by which knowledge can be derived. The absolutist-relativist dimension is concerned with the status of knowledge.

An extreme absolutist believes that there is an objective truth about the world which it is possible to discover. For an extreme absolutist, knowledge has an objective existence independent of the knower. It follows that given any candidate for knowledge, it is possible to determine in an absolute sense whether this knowledge is true. In terms of the class-instance distinction, the extreme absolutist believes that general knowledge, in the form of a class, has objective existence and it is possible to determine those classes. Extreme absolutism is represented diagrammatically in Figure 1.3.

Extreme absolutism can be summed up by the slogan 'Reason rules the world' attributed to Anaxagoras a pre-Socratic philosopher [Leahey 87].

In contrast, for the extreme relativist, there is no such thing as objective knowledge. Given two different groups or individuals, it may be impossible to make any comparison of their different knowledge, since even the
particular knowledge of the groups might be distinct. Extreme relativism is represented graphically in Figure 1.4:

Extreme relativism can be summed up by the slogan 'Man is the measure of all things' attributed to Protagoras a sophist and contemporary of Socrates [Leahey 87].

As for rationalism and empiricism, extreme absolutism and extreme relativism represent opposite poles along a absolutist-relativist dimension. Different epistemologies are situated along the dimension according to how much they consider different forms of knowledge are comparable.

Any framework which supports formal language design will, of necessity, have an underlying epistemology. The reason for introducing the two dimensions of epistemology is to enlighten the discussions of Chapters 2 and 3 which establish this epistemology.

Two dimensions are necessary, since it is essential for the purposes of the thesis to separate the concept of rationalism from the concept of absolutism. Often, when the term rationalism is used, it is employed to mean a combination of rationalism and absolutism as defined above. To justify such a separation, it is necessary to present a rationalist relativist viewpoint.
Consider the definition of extreme rationalism presented above:

'Extreme rationalism can be defined as the belief that it is possible to obtain knowledge by the means of reason alone.'

The key term in the definition is 'reason'. If reason is considered an absolute, that is there is only one valid form of reason, then there is only one valid form of knowledge, and thus we have an absolutist position. However, if on the other hand, reason is relative, that is accepted by a group, then the knowledge that is derived from that reason is knowledge accepted by that group. Further, if reason is even more relative, that is what is accepted by a group with respect to some purpose, then the knowledge derived from that reason is what is accepted by the group with respect to that purpose.

It turns out that the notion of formality, and the corresponding notion of formal language, developed in this thesis constitutes an example of rationalist relativism. Since the philosophical position taken by the thesis is essentially relativist, the terms rationalism and formalism become interchangeable. Thus, the rational aspects of the framework correspond to the formal aspects of the framework. When discussing the conception and the framework, the term formal is used. However, to avoid confusion, when discussing different philosophies of mathematical and scientific knowledge the term rationalism is employed.

1.4 Overview of the PhD

As has already been stated, the class-instance distinction outlined earlier has been used as a device for structuring the thesis at the macro level.

The first four chapters attempt to establish a conception of formal language design. This conception acts as a specification for a framework to support formal language design. Thus, the conception represents a class whose instances are possible frameworks. The partial development of a framework in line with the conception is presented in Chapter 5 onwards. An overview of these two components of the thesis is presented in the following two sections.
1.5 The Conception of Formal Language Engineering

The first part of the thesis, Chapters 2 to 4 are concerned with establishing a conception of formal language design. As has already been stated, this conception acts as a specification for the framework developed in the later part of the thesis.

Chapter 2 considers the nature of formality, by considering different philosophies of mathematics, outlining their position along the dimensions of epistemology. It illustrates how these different philosophies give rise to different notions of formality. It considers different definitions of formality given in common text books and finally presents a strongly relativistic definition of formal language.

Chapter 3 considers the nature of disciplines, culminating in the definition of a generic engineering conception of design disciplines, of which the conception of formal language design is an instance. It considers approaches to the philosophy of science, outlining their position along the dimensions of epistemology. The relationship between science and design is considered in the light of work considering this issue in the discipline of HCI [Long & Dowell 89]. It examines the relationship between formality and design, by focusing on philosophical debates concerning formal methods in computer science. Finally, the generic engineering conception is outlined and related to the previous sections.

In Chapter 4, the generic engineering conception of Chapter 3 is instantiated for the design of formal languages. The conception that results is based strongly upon work that aims to establish a conception for HCI [Dowell & Long 89]. The conception introduces the notion of a formal language worksystem, which achieves work benefits and incurs worksystem costs. The desired performance of a work system is expressed as a combination of desired worksystem costs and desired work benefits.
1.6 The Framework for Formal Language Design

The second major part of the thesis, Chapters 5 to 11 outlines a partially developed formal framework to support the design of formal languages.

Chapter 5 outlines the basis of the formal framework. It justifies the choice of notation for expressing the framework, the Z notation enriched with category theory. It then outlines the basic constructs of Z and specifies the key concepts of category theory within Z. The examples presented in this section are built upon in later chapters.

Chapter 6 defines the notion of performance requirements of a formal language worksystem. The performance requirements are constructed from the cost requirements and the benefit requirements of a worksystem. The cost requirements are expressed in terms of cost structures and the benefit requirements are expressed in terms of a domain of work transformations. The performance requirements for a simple paint shop are specified to illustrate, and validate, the performance requirements definition.

Chapter 7 presents an extensive specification of benefit requirements for an air traffic control system. The aim of this chapter is to provide a medium-sized validation of the benefit requirements definition.

Ideally, Chapter 8 should define the notion of a formal language worksystem specification. In the time period of the thesis, it has not been possible to develop a general formal definition of the concept of a formal language worksystem. This section presents a definition of a particularly restricted class of formal language worksystems. The definition of the restricted class is illustrated with a specification of a paint shop worksystem which satisfies the paint shop performance requirements outlined in Chapter 6. The remainder of the chapter considers how to achieve a general and formal characterisation of the notion of a formal language worksystem.

Chapter 9 considers a particular aspect of formal language worksystems, the definition of formal language syntax. It considers a number of syntactic formalisms, ranging from text based grammars for specifying programming language syntax to shape grammars for expressing architectural designs. It
establishes a framework in which all the grammars can be expressed. In other words, it establishes a class of which all the grammars are instances. The framework is then used as the basis for constructing a novel grammatical formalism.

Chapter 10 presents an extensive specification of the syntax of a notation used for describing the rationale behind a design [MacLean et al. 91]. The aim of this chapter is to provide a medium-sized validation of the framework for syntax definition presented in Chapter 9.

Chapter 11 discusses the notion of an engineering principle. It considers the key components of a principle, and how principles might be developed. The issue of the status of a principle, and the nature of the guarantees it offers is discussed.

Chapter 12 presents the conclusions concerning the work of the PhD. The conclusions are based upon the criteria for assessment outlined in the following section.
1.7 Criteria For Assessment

While incrementing knowledge is part of the work of this thesis, it aims primarily to create a synthesis of work across a broad range of disciplines.

There are four criteria by which the work may be judged. These are:

- Internal consistency between conception and framework
- External synthesis at the level of the conception
- External synthesis at the level of the framework
- Knowledge incremented at the level of the framework

Assessing the internal consistency between the conception and the framework requires an answer to the question:

How well does the formal framework outlined in the second part of the thesis instantiate the conception presented in the first part?

Assessing the external synthesis at the level of the conception requires an answer to the question:

How useful is the conception as an approach to the design of complex systems?

This question can be asked both at the level of the generic engineering conception and at the level of the conception as instantiated for formal language design.

Assessing the external synthesis at the level of the framework requires an answer to the question:

How useful is the framework in supporting the design of complex systems?

This question can be asked of the framework as a whole and of the individual components of the framework.
Assessing the knowledge incremented at the level of the framework requires an answer to the following question:

How much do the individual components of the framework increment existing work?

The conclusion in Chapter 12 will attempt to answer all of these questions.

1.8 Conclusion

This chapter has established the personal motivation for the thesis. It has outlined the class-instance distinction, which is used as both a macro and micro level structuring mechanism throughout the thesis. It has developed the rationalist-empiricist and absolutist-relativist dimensions of epistemology, which are used in the early chapters of the thesis in the discussions of the nature of formality and the nature of disciplines. An overview of the chapters of the thesis has been presented. Finally, the criteria by which the thesis is assessed in the conclusion are presented.

The next chapter discusses the nature of formality.
2 On the Nature of Formal Languages

The purpose of this chapter is to make clear what is meant in this thesis by the term Formal Language.

The chapter contains the following sections:

- Philosophy of Mathematics
- The Notion of Formal Language

The first section considers how different notions of formality are dependant upon an underlying philosophy of mathematics. Thus, the first section illustrates the need to be clear about what is meant by a formal language. The second section employs the philosophical considerations of the first section to develop the notion of formal language as used in the thesis.
2.1 Philosophy of Mathematics

'The current predicament of mathematics is that there is not one but many mathematics and that for numerous reasons each fails to satisfy the opponents of the opposing schools. It is now apparent that the concept of a universally accepted and, infallible body of reasoning - the majestic mathematics of 1800 and the pride of man - is a grand illusion. Uncertainty and doubt concerning the future of mathematics have replaced the certainties and complacency of the past. The disagreements about the foundations of the 'most certain' science are both surprising and, to put it mildly, disconcerting. The present state of mathematics is a mockery of the hitherto deep-rooted and widely reputed truth and logical perfection of mathematics.' [Klein 80]

This section introduces different philosophies of mathematics and illustrates how the different philosophical positions give rise to different conceptions of what is meant by a formal language. The different philosophies considered here arose at around the turn of the century. By and large, they resulted from problems that had begun to arise in mathematics a century earlier, as the quote above illustrates.

By the dawn of the 19th century, empiricist philosophers such as Hume [Hume, A Treatise on Human Nature], and rationalist philosophers such as Kant [Kant, Critique of Pure Reason] had begun to doubt the existence of a knowable absolute truth. However, mathematics, the field that many believed to be the epitome of rationalism, was in its heyday. While there may be problems for metaphysics, mathematical understanding of the world was increasing at a rapid pace. What is more, mathematical knowledge was often equated with knowledge of the truth.

One of the strongest pieces of evidence to support the hypothesis of 'Mathematics as Truth' was Euclidean Geometry. Euclid established axioms for geometry in 300BC. From the relatively few, 'intuitively obvious', axioms it was possible to derive the theorems about the whole of geometry.

The fact the axioms were 'intuitively obvious' was not really open to much question. It was, however, true that one axiom, the parallel postulate, was slightly worrying, and from 300BC to 1800 many attempts had been made to
derive it from the others. The parallel postulate essentially states that two parallel lines will never intersect.

In the early 19th century, a number of mathematicians established geometries in which the parallel postulate was assumed to be false. The new geometries raised two questions:

- Which geometry was the 'true' geometry of the 'universe'?
- Are the axioms of any of the possible geometries, including Euclidean, consistent? In other words, how was it known that the axioms of any geometry did not lead to contradictions?

Before the development of non-Euclidean geometries, the axioms of geometry were felt by rationalists to be a prime example of an a priori truth. The first question opened up what was thought to be a priori truth to judgement by perceived phenomena. Indeed, the geometry of space time proposed by Relativity theory is non-Euclidean. Many physicists believe Relativity theory bears a closer resemblance to observed phenomena than the Newton view, at least on the scale of the cosmos. Thus, non-Euclidean geometries were clearly a blow to the rationalist conception of the truth. It can also be seen as a blow for absolutism. The invention of the non-Euclidean geometry perhaps led many mathematicians to cease working in the field of perceived phenomena - creating a schism between pure and applied mathematics.

The second question, that of consistency, sparked off a search for a solid foundation for mathematics. It was intended that the foundations would be able to demonstrate their own consistency. The search proved fruitless. In the 1930s, it was demonstrated by Godel that any system which was powerful enough to include arithmetic could not prove its own consistency.

Till the 1800s and beyond, the dominant philosophy of mathematics was Platonism. The crisis in the foundations gave rise to two other viewpoints: constructivism and formalism. These three approaches to the nature of mathematical knowledge, and the notion of formal language that arises from each approach, are detailed in the next three sections.
2.1.1 Platonism

'Most writers on the subject seem to agree that the typical working mathematician is a Platonist on weekdays and a formalist on Sundays. That is when he is doing mathematics he is convinced that he is dealing with an objective reality whose properties he is attempting to determine. But then when challenged to give a philosophical account of this reality, he finds it easier to pretend that he does not believe in it after all' [Davis & Hersch, 81]

According to the Platonist, mathematical objects are objective facts which have an existence independent of our knowledge of them. The process of doing mathematics involves discovering objects which represent mathematical truth. Mathematical demonstrations should be believed, because they are based upon undeniably true principles.

As the above quote indicates, the Platonist position is difficult to defend. Key reasons for this difficulty arise in the historical developments outlined above. The 'obvious truth' of the Euclidean axioms of geometry being a case in point.

The advent of non-Euclidean geometries weakened the Platonist position, but did not destroy it. If a consistent mathematical system could be found in which all the geometries could be embedded, then paradise would be regained. Towards the end of the 19th century, such a foundation was proposed based upon various theories of sets. These early theories of sets, however, were found to be inconsistent, and the only way of avoiding the inconsistencies involved assuming what many believe to be non-intuitive principles.

There seemed no reason to assume that the non-intuitive principles would necessarily be consistent themselves. So, attempts were made to prove the consistency of these principles. These attempts culminated with Godel's results in the 1930s that demonstrated that such proofs were, in general, impossible to obtain.

The mathematical viewpoint of Platonism epitomises rationalist absolutism. Indeed the 'obvious truth' of mathematical assertions, such as Euclid's axioms
of geometry, were used as arguments in favour of the absolutist position of an *a priori* concept of truth.

The term Platonism derives from the philosopher Plato's concept of the Forms. At different points in his lifetime, Plato offered different explanations about how the Forms were knowable. One strand of his thought equated the true Forms with the objects of mathematics. Thus, a mathematical triangle represents a true Form of a triangle. It is from this strand of thought that mathematical Platonism developed.

Thus, for a Platonist mathematician the formal is equated with the true. A formal language is a language of truth.

### 2.1.2 Constructivism

'Mathematics belongs to man, not to God. We are not interested in properties of the positive integers that have no descriptive meaning for finite man. When a man proves a positive integer to exist he should know how to find it. If God has mathematics of his own that needs to be done, let him to it himself' [Bishop, 67]

'All schools of constructive mathematics reject the notion of an *a priori* concept of truth.' [Beeson 80]

For the constructivist, mathematics is invented rather than discovered. Mathematics can be viewed as originating as constructions in the mind of the individual. Mathematical demonstrations are to be believed, because they are based upon intuitively believable principles, i.e. principles accepted by all those party to the demonstration.

Since for the constructivist, there is no *a priori* notion of truth, it is meaningless to talk of truth or falsity. Consequently constructivists reject the principle of the excluded middle, that given any proposition X:

- either X or not X

It was the rejection of the principle of the excluded middle, and the belief that this rejection meant that abandonment of many areas of mathematics which
depended on it for their proof, that led to the formation of the third mathematical position, that of formalism.

If Platonism epitomises rationalist absolutism, then constructivism, or sometimes intuitionism, epitomises rationalist relativism. As the quote of the previous section indicates, constructivists form a minority of mathematicians. Most of these mathematicians work in highly abstract fields of mathematical logic.

For a constructive mathematician, a Formal Language is not a means for expressing the truth, but rather a convenient device for communicating mental mathematical constructions.

2.1.3 Formalism

'No one, though he speak with the tongues of angels, will keep people from...using the principle of excluded middle.' Hilbert quoted in van Heijenoort [van Heijenoort 67]

'The belief in the universal validity of the principle of the excluded third in mathematics is considered by the intuitionists as a phenomenon of the history of civilisation of the same kind as the former belief in the rationality of $\pi$, or in the rotation of the firmament about the earth.' [Brouwer, 81]

Formalism can be viewed as an attempt to rescue mathematics from the perceived horrors of constructivism. For a formalist, mathematics can be considered as a game played with symbols which have nothing to do with objective reality.

For some, notably the French school collectively referred to by the pseudonym of Bourbaki, formalism is a weakened Platonism. From its point of view, inconsistencies will inevitably arise in any formal system. From these inconsistencies the system is revised and more is learnt about mathematics. Through inconsistencies formal mathematical systems evolve, moving closer and closer towards the mathematical truth. The Bourbaki view is very similar to a view of the nature of knowledge, that is expounded by the philosopher of science Lakatos (see Chapter 3).
As the quote at the beginning of the subsection on Platonism indicates, most mathematicians today would consider themselves formalists not due to any strong philosophical commitment, but rather to the belief that much of their subject area would be lost if the law of excluded middle was rejected. This view was held by the reputed founders of both constructivism and formalism Brouwer and Hilbert. In fact, Bishop's work on Constructive Analysis [Bishop 67] indicates that:

'...both Hilbert and Brouwer had been wrong about an important point about which they had agreed. Namely, both of them had thought that if one took constructive mathematics seriously, it would be necessary to 'Give Up' the most important parts of modern mathematics' [Beeson 80]

In many ways, formalism maintains a very broad position by denying that mathematics is anything other than a game with symbols. Formalism can accommodate positions from rationalist relativism to rationalist absolutism. However, most formalists would like to think of mathematics as an objective truth which their activities discover.

For the formalist, mathematics is just the manipulation of a formal language. The formal language is merely a collection of strings formed from a collection of symbols.

2.1.4 Conclusion

It can be seen that different philosophical perspectives on mathematics give rise to different notions of what is meant by formal.

At the absolutist end of the spectrum, for what might be called the naive Platonist, formal languages are a means of expressing the objective truth. The objective truth of the assertions in the language is understood by individuals a priori. An interesting consequence of the naive Platonist position, is that when an individual fails to understand a statement in the formal language, it is the individual who is in error not the language. The whole notion of
designing formal languages is at odds with the Platonist notion of formal language.

Historical evidence tends to refute the naive Platonist position. Consequentially it is reformulated in a weaker position, that of Bourbaki, one that might be called the Platonist-formalist position. Formal languages are still the means of expressing the objective truth but, now, the formal language cannot represent the truth perfectly. Indeed, it is the failure of the formal language to express the truth that gives rise to inconsistencies. The inconsistencies are resolved by redesigning the formal language. In contrast to naive Platonism, from the Platonist-formalist position, formal language design is now at least an issue. The problem is that at any point in time there is only one formal language, the current version of objective truth. This language will only be redesigned to remove inconsistencies and thus move closer to objective truth.

To the pure formalist, mathematics is a game played with symbols, and it is these symbols that make up a formal language. Any relationship between the symbols and the truth has been abandoned. To the pure formalist, if formal languages happen to be useful, then usefulness is incidental. This approach has, in a sense, attempted to resolve the dilemmas of Platonism by basing the whole of mathematics upon the notion of a formal language. In contrast to the Platonist, for the formalist it makes sense to define what is meant by a formal language. For the formalist, however, since the utility of the language is not at issue, the definition of formal language is a narrow one, concerned only with strings of character symbols.

Finally, to a constructivist, mathematical objects are mentally constructed by the individual. A formal language is a means of expressing these mental constructions.

'We may consider the formal system as the linguistic expression, in a particularly suitable language, of mathematical thought.' [Heyting, 76]

Thus, for a constructivist mathematician, the design of formal languages is clearly an issue, since different formal languages may facilitate the communication of different forms of mental construction.
To recap, the aim of the thesis is:

To develop a framework to support the design of formal languages.

It seems that a constructive philosophy of mathematics is almost a consequence of this aim. At least, it appears to provide the only perspective on formal languages that is not inconsistent with the aim. What is required, however, is a redefinition of the concept of formal language as a means for expressing mental constructions. This redefinition is developed in the next section.
2.2 The Notion of Formal Language

For some authors the formal is to be equated with the mathematical. For example, consider the following definition which appears in an Open University Software Engineering text:

"By formal language we mean a language with a mathematically defined (and hence precisely defined) syntax and semantics" [OU 90]

However, considering the different philosophies of mathematics outlined in the previous section, such a definition carries with it certain ambiguities. This section discusses other authors' attempts to define the concepts of language and formality in the light of the philosophical discussions of the previous section. Then, in the light of these discussions, it establishes the definition of formal language that is employed throughout the thesis.

Although in what follows, various definitions of formality are classified according to whether they typify a formalist or a constructivist viewpoint, it is not intended to claim that the authors of these definitions necessarily hold the particular philosophical position, merely that their definition fits well with a particular position.

2.2.1 Formalist Formality

Recall that formalism considers mathematics to be a game with symbols. It is not surprising then, that a formalist approach tends to equate formality with the syntax. Consider, for example, the following definition:

"A formal language comprises two parts, its alphabet which specifies what symbols are to be found in the language, its syntax which specifies how these symbols may be put together" [Woodcock & Loomes 88]
In the same manner the following definition from an introductory logic textbook equates formalisation with a separation from meaning:

"To formalise is to strip away the concepts which give meaning and application to the subject, so that nothing remains but the bare symbols" [Hodges 85]

Formality without meaning arises due to a formalist view of mathematics. The problems that arose with the foundations were problems caused by the meaning of mathematical statements. To overcome these problems, the formalist has to separate the formal aspect of a language from its meaning. The separation of meaning from formality can be seen as an attempt to remove controversy from language. In an absolute sense, it is impossible for the concept of a formal language to include meaning, and the resultant formal languages to be non-controversial. Since ultimately everything can be disputed. However, it is possible for the concept of a formal language to include meaning if it merely seeks to be relatively non-controversial. That is, non-controversial to a certain group. It is such a definition of formality that is consistent with a constructivist philosophy.

### 2.2.2 Constructivist Formality

In his 'Introduction to Knowledge Base Systems, Frost observes that the importance of formality is in enabling communication:

'Formal languages and notations for the representation of knowledge are important since they enable the resulting representations to be interpreted correctly by people other than those that encoded them' [Frost 86]

This link between formality and communication is clearly in line with the views of the constructivist Heyting quoted in the last section:

'We may consider the formal system as the linguistic expression, in a particularly suitable language, of mathematical thought.' [Heyting, 76]
It is perhaps then not surprising to see that Frost's definition of formality includes both the syntax and meaning:

'By 'formal' we mean that the language is well-defined in the sense that (a) rules exist for the construction of legal expressions and (b) rules exist such that the meaning of legally formed expressions can be defined from the meaning of the components of those expressions' [Frost 86]

It is interesting, that although Frost considers the importance of formal languages lie in their ability to aid communication, the definition of formality makes no mention of communication. Frost's definition maintains the form of typical definitions of formality, but his interest in the utility of formality, which is similar to Heyting's view of the suitability of formality, begins to present an alternative approach to the defining formality. An approach which might concentrate on providing a more empirically orientated definition.

Two authors from different backgrounds have advocated a more empirically based approach to issues of formal language, although both employ the term notation.

2.2.3 Goguen and the Social Aspects of Notation

Goguen argues in the paper 'On Notation' [Goguen 93a], for the importance of evaluating notation. Notation is defined:

"... in a broad sense that includes the design of 'icons', screen layout, colour, motion and interaction, as well as choice of linear syntax and keywords" [Goguen 93a]

Goguen still, however, equates formality and text as the following quote from the same paper indicates:

"I argue for the importance of notation, and in particular, for the importance of diagrams and other visual forms of presentation, as opposed to purely formal, textual representations." [Goguen 93a]
In the bulk of 'On Notation', Goguen employs the work of C. S. Pierce's semiotics to attempt to demonstrate the importance of the social issues in relation to notation, advocating the approach of Ethnomethodology as a means for evaluating notations. The paper concludes:

"It follows that as designers of languages and tools, we should try to discover the categories and methods of our user communities, and base our designs on them, rather than trying to impose our own categories and methods from afar." [Goguen 93a]

Goguen continues this theme in another paper, which argues for a 'Social Theory of Information':

"Such a theory of information would have to take full account of social context, including how information is processed and used, rather than merely how it is represented; that is it must be a social theory of information, not merely a theory of representation." [Goguen 94]

The same paper concludes with a definition of information:

"as an interpretation of some configuration of signs for which some social group may be held accountable." [Goguen 94]

The implication here is that information is dependant not just upon signs but also upon the agents that use it.

2.2.4 Green and the Cognitive Aspects of Notation.

According to Green the environment in which notation is used is of central importance in any analysis of notation:

"Indeed the relationship between notation and the environment is such that the notation cannot be used except in some kind of environment of use." [Green 89]

This is embodied in the slogan:

"System = Notation + Environment" [Green 89]
Like Goguen, Green uses the term 'notation' in the same manner as the thesis employs the term 'language'. Further to the description of system in terms of notation and environment, Green observes:

"Each notation highlights some types of information at the expense of obscuring other types; each notation facilitates some operation at the expense of making others harder. A notation is never absolutely good, therefore, but good only in relation to certain tasks" [Green 89]

Thus, for Green, notations cannot be considered apart from some environment and can be only evaluated in relation to some task.

2.2.5 Conclusion - Defining Formality

Section 2.1 concluded that a constructivist philosophy of mathematics is almost a direct consequence of considering the design of formal languages. This section has explored different authors' definitions of, and approaches to, the concept of formal language. From a constructivist perspective, one can identify three components that are necessary to form the definition of the term formal language. These are:

Language
Agent
Task

Language can be thought of as any means of communication. The aim of the definition is to include, for example, collections of text, graphics, sound waves, electromagnetic waves, etc. The notion of language is now sufficiently broad to enable the design of as broad a range of languages as possible. The concept of language as used here is equivalent to the term notation as employed by both Goguen and Green.

An agent may be thought of as an entity which employs or interprets language. For a constructivist, language is a means of expressing mental constructions. An agent employs language in order to express mental constructions, and interprets language in order to understand others' mental constructions. An agent may be a human user of a language, and therefore a
set of agents might form, what Goguen termed, a social group, that might be held accountable for a language. An agent may be a machine which interprets and employs language, and therefore an agent, or a set of agents may form, what Green terms, an environment.

A task is the purpose for which language is employed and interpreted by agents. The concept of task provides the means for overcoming the formalists problems with meaning. Meaning can be interpreted as expressing the relationship between a language and the task which it was designed to support. As pointed out by Green, languages can only be evaluated with respect to some task. Evaluating a language with respect to a task amounts to determining whether the language can be employed and interpreted by the agents, so that the task is carried out.

Putting the concepts of language, agent and task together, it is now possible to establish the definition of formal language employed in the thesis:

A language is formal with respect to a class of agents, and a task, if it can be interpreted and employed by those agents so as to carry out the task.

This definition is further elaborated on in Chapter 4 when the Conception of Formal Language Engineering is outlined in detail.
2.3 Conclusion

This chapter has considered three main perspectives from the philosophy of mathematics; Platonism, formalism and constructivism. It has examined the different conceptions of formal language that arise with each philosophy and demonstrated that only a constructivist philosophy is consistent with formal language design concerns.

Other authors' definitions of, and approaches to, formal language have been considered. From these considerations, and in line with a constructive philosophy, the basic concepts of language, agent and task were introduced. These were then used to outline the definition of formal language employed in the rest of the thesis.

This chapter has considered the philosophy of mathematics in order to develop the concept of formal language employed in the thesis. The next chapter considers the philosophy of science in order to develop a view of the nature of a framework to support design.
3 On the Nature of Frameworks

The aim of this thesis is:

To develop a framework to support the design of formal languages.

This chapter works towards developing a conception of what such a framework should be. The first section considers various different conceptions of the frameworks of scientific disciplines that have been outlined by philosophers. The second section considers three different conceptions of frameworks (craft, applied science and engineering) that have been proposed for the design discipline of HCI. The third section builds upon the work of sections one and two to establish a generic conception of a framework to support engineering design. The fourth section exploits the generic engineering framework to resolve some framework related issues that have arisen in the field of software engineering.
3.1 Philosophy of Science

This section considers various different conceptions of frameworks of scientific disciplines. This section includes the following subsections:

- Naive Inductivism
- Naive Falsificationism
- Lakatos's Research Programs
- Kuhn's Paradigms and Disciplinary Matrix
- Feyerabend's Anarchistic view of knowledge

The list is in many respects far from complete, yet it provides a broad sample across the absolutist-relativist and rationalist-empiricist dimensions of epistemology.

3.1.1 Naive Inductivism

Naive inductivism proposes that scientific knowledge is constructed from the 'bottom up'. By this, it is meant that theories about the world are generalised from phenomena using the process of induction (See Figure 3.1).

![Figure 3.1 The naive inductivist framework](image)

Having seen one thousand swans, all of which are white, one might infer, using induction, the general theory that:

All swans are white.
Naive Inductivism suffers from two main problems:

- Uncertainty of Inductively Derived Truths
- The Theory Dependance of Observation

The uncertainty of inductively defined truth has been known since the time of classical Greece. Swan one thousand and one may turn out to be black, in which case the scientific knowledge concerning swans amounts to nothing.

The problem of the theory dependance of observation arises, since theory is, in itself, a valuable tool in guiding observation of phenomena. The belief that all swans are white might enable one to determine that the large white bird in the middle of the lake is a swan. The observation of swans is dependant upon the theory about swans. In a similar fashion, a black swan in the middle of the lake, a phenomenon which would refute the white swan theory, might not be identified as a swan, because of the swan theory.

Some forms of naive inductivism, notably that of Comte, epitomised empiricist absolutism in the same way that the mathematical Platonism epitomises rationalist absolutism. Comte believed that from empirical observations irrefutable scientific generalisations could be constructed. In modern times inductivism is not seriously considered, by any, as a valid conception of the scientific process. It is most commonly thought of as one of many methods used in the process of the construction of scientific knowledge.

3.1.2 Naive Falsificationism

In response to the problems of uncertainty of inductive truth, the perspective of falsificationism was developed. Naive falsificationism proposes that scientific knowledge is constructed from the 'top down' (see Figure 3.2). Bold theories are outlined, then a search is undertaken for phenomena which refute the theory.
From this perspective, science does not grow, but rather evolves. Once the theory that all swans are white is formed, scientists search relentlessly for a counter example. Having finally reached the southern hemisphere and found a black swan, the first theory is refuted. Almost immediately a new theory is born, this time:

All swans from the northern hemisphere are white.

The process of searching for a counter example begins again.

Falsificationism, although accepting the uncertainty of scientific knowledge, proposes that knowledge can move towards certainty. In addition, falsificationists propose that a concept is scientific to the degree that it is falsifiable. To some, falsificationism is a politically useful tool, since schools of thought whose concepts are not falsifiable can be rejected as un-scientific.

However, the problem of theory dependance of observation remains for the falsificationist. Suppose for example another swan theory existed:

All swans live in the northern hemisphere.

Since this theory was not the theory under consideration, it could be used to help guide the search for the counter-example phenomenon saving needless searching in the southern hemisphere. The theory dependance of observation calls into question whether the knowledge generated by falsificationist science can move towards certainty. It also calls into question the falsificationist claims that true science is falsifiable, since any demonstration
of the degree of falsifiability of a concept is dependant upon existing concepts.

In many ways naive falsificationism is both less absolutist than naive inductivism and more rationalist. Hypotheses are constructed with reason and falsified empirically. Hypotheses don't represent absolute truth but a stage on the way to truth.

3.1.3 Lakatos's Research Programs

'The first world is the material world, the second world is the world of consciousness, the third world is the world of propositions, truth, standards: the world of objective knowledge' [Lakatos 74]

'One cannot understand the history of science without taking into account the interaction of the three worlds' [Lakatos 74]

The perspective of naive inductivism and the perspective of naive falsificationism treat individual and collective knowledge as a single entity. No account is taken of the fact that science is rarely an individual exercise, but more commonly involves many individuals at many different times and places. The next two sub-sections describe conceptions of frameworks of scientific disciplines which attempt to take account of the social nature of science.

The first framework outlined is Lakatos's 'Methodology of Research Programs'. Largely based upon the work of Popper, the methodology of research programs really amounts to a more sophisticated falsificationist viewpoint. As can be seen from the above quotations it is also strongly in line with an absolutist conception of knowledge.

Lakatos uses the term research program to describe a scientific discipline. In Lakatos's view a research program may be conceived as a framework which contains the following elements:

- Positive Heuristic
- Negative Heuristic
- Hard Core
A Set of Auxiliary Hypotheses

The relationships between these elements is seen in Figure 3.3:

The hard core consists of the central concepts of the theory. The negative heuristic banishes attempts to falsify the hard core. The auxiliary hypotheses consist of a protective belt of hypotheses which are extensions to the theory to take account of anomalies and predict novel phenomena. The positive heuristic indicates what are acceptable auxiliary hypotheses, and hence is a generalisation of auxiliary hypotheses concepts.

So far, little has been mentioned about truth, a concept of central importance to any absolutist. Lakatos attempts to deal with the concept of truth by talking about progressive and degenerate research programs. To consider what is meant one needs to consider how research programs change over time. If, over time, the number of novel phenomena a research program predicts increases, then the research program is said to be progressive. If the number of novel phenomena decreases, then the research program is said to be degenerate.
In many ways, Lakatos's methodology of research programs is essentially a sophisticated falsificationism which takes account of science as a social enterprise. As such, Lakatos occupies the same position with respect to the Rationalist-Empiricist and Absolutist-Relativist dimensions.

3.1.4 Kuhn's Paradigms and Disciplinary Matrix

'Scientific knowledge, like language, is intrinsically the common property of a group or else nothing at all. To understand it we shall need to know the special characteristics of the groups that create it and use it.' [Kuhn 70]

Kuhn's concept of a scientific discipline is in many ways very similar to Lakatos's methodology of research program. This similarity concerns the framework of a scientific discipline. Where the two authors differ is on the status of the knowledge, Kuhn taking a significantly more relativistic stance.

In Kuhn's view, a scientific discipline may be conceived of as a framework which contains the following elements:

- Paradigm
- Disciplinary Matrix
- Shared Exemplars

The relationships between these elements is expressed in Figure 3.4.

For Kuhn, the history of a scientific discipline cycles through two distinct phases.

The shortest phase is the crisis period. During this period the symbolic generalisations, metaphysical assumptions and system of values that form the disciplinary matrix are in question. Rival positions abound until one begins to dominate. At this point, the discipline moves into a period of normal science.

During normal science the scientific community holds a consensus view concerning the disciplinary matrix. Scientists solve scientific problems within
the paradigm of shared exemplars. The shared exemplars, consisting of theory predictions, are developed until such time as it becomes increasingly difficult to develop exemplars whose theory predictions accord with perceived phenomena. At this point, a period of crisis ensues.

The conceptions of science proposed by Lakatos and Kuhn are indeed very similar. Where they differ is on the question of how a scientific discipline might be assessed. Lakatos believes that it is possible to assess how well the discipline approximates to the absolute truth depending upon whether it is progressive or degenerate. Kuhn believes it is possible to assess the discipline depending upon whether it is useful in the solving of problems. The term 'useful in the solving of problems' may be thought of in a variety of ways. Work in the field of thermodynamics might be of useful in the solving of problems in the design of chemical plants, whereas work in high energy particle physics may be useful in the solving of the problem of satisfying a society’s desire to develop a model of the world.
On the Rationalist-Empiricist dimension, the work of Kuhn and Lakatos is roughly equivalent. Kuhn's approach is, however, significantly more relativistic than that of Lakatos.

3.1.5 Feyerabend's Anarchistic view of knowledge

Feyerabend's strongly relativist views might be considered as the antithesis of the position of Lakatos. Feyerabend's position concerning science is given reasonably clearly in the title of his book 'Against Method: An Anarchistic Theory of Knowledge'. Feyerabend denies:

'...that there can be a theory of science and of knowledge.' [Feyerabend, 87]

For Feyerabend, science is a process dominated by creativity. He rejects any notion of a theory of science, because it would act to restrain the process of science hampering creativity. Any theory of science, he believes, is in danger of becoming an orthodoxy to which science must correspond.

Thus, it may appear that Feyerabend would be opposed to the central aim of this chapter, that is to develop a conception for a framework to support language design. However Feyerabend does claim that:

'...scientists are not only responsible for the correct application of standards they have imported from elsewhere, they are responsible for the standards themselves. Not even the laws of logic are exempt from their scrutiny, for circumstances may force them to change logic as well...' [Feyerabend, 87]

The views of Feyerabend act to clarify the purpose of the framework to support language design. The purpose of explicit postulation of a conception is not to impose constraints on those wishing to design formal languages. It is merely to make explicit the concepts that have contributed to the development of a framework to support language design.
3.1.6 Conclusion

The aim of this section was to consider the descriptive conceptions proposed by philosophers of science as an aid to the development of a conception of a framework for formal language design.

The frameworks of Lakatos and Kuhn are structurally very similar. The generic conception of artifact design outlined in the conclusion to this chapter will also have similar components. Kuhn, however, differs from Lakatos about the nature of knowledge, taking a significantly more relativistic stance. To Kuhn, knowledge has value, if it can be used in solving problems, not according to some criteria of objective truth. The situation in the philosophy of science can be seen in many ways as similar to that in the philosophy of mathematics presented in Section 2.1.

Indeed, the path from naive inductivism, to falsificationism, to Lakatos methodology of research programs, can be seen as a retreat from the empiricist-absolutism of Comte that parallels the retreat from Platonist rationalist-absolutism. Kuhn and Feyerabend and their relativist stance can be seen as an empirical equivalent of mathematical constructivism.

As mathematical constructivism was a useful notion in developing the definition of formal languages, so Kuhn's notion of knowledge that is 'useful in the solving of problems' shall be key in developing the generic conception of artifact design. The conception will, however, go further than Kuhn, detailing how to assess the utility of knowledge to support design.
3.2 Science and Design

This section considers some different possible conceptions that have been considered for the discipline of HCI [Long & Dowell 89]. In the paper, Long and Dowell, consider three components of a discipline:

- Knowledge
- Practice
- General Problem

Knowledge is used to support practice aimed at solving the general problem of a discipline. For the discipline of HCI the scope of the general problem is identified as:

‘humans and computers interacting to perform work effectively’. [Long & Dowell 89]

The scope of the general problem for HCI is represented in Figure 3.5.

![Figure 3.5 The Scope of the General Problem for HCI (taken from [Long & Dowell 89])](image)

The concept of the general problem of HCI is extended in another paper [Dowell & Long 89]. The human and computer interacting together are thought of as an interactive worksystem. The concept of effective work is captured by the notion of desired worksystem performance, which is expressed in terms of both the desired quality of work and the acceptable costs of the human and computer components of the worksystem, that are incurred in doing the work. Interactive worksystems exhibit actual
performance, which is a function of the actual quality of work done by the
worksystem and the actual costs incurred. A worksystem can be said to be
effective if actual performance corresponds to desired performance.

Long and Dowell [Long & Dowell 89] identify three different conceptions of
the discipline of HCI that are distinguished by the different nature of their
knowledge and practices. These conceptions are:

Craft
Applied Science
Engineering

In what follows, each of these conceptions is outlined turn. The nature of the
knowledge and practices that correspond to each conception are considered
in terms of their definition, operationalisation, testability and generalisation.

The term definition is employed to mean the explicit description of the
knowledge and practices. Operationalisation is the transformation of the
definitions of the knowledge and practices into a form that can be used and
tested. The testing of knowledge and practices is aimed at determining how
well they support the general problem. Finally, knowledge and practices are
general if they can be applied to more than one instance of the general
problem.
3.2.1 Craft Conception

The conception of a craft design discipline of HCI describes a practice in the form of 'implement and test' and knowledge in the form of 'heuristics'. In the craft discipline, artifacts are designed by a process of construction and evaluation and reconstruction guided by the use of informal heuristics. The conception of HCI as a craft discipline is represented in Figure 3.6.

By considering examples of HCI as a craft discipline, Long and Dowell conclude that although the knowledge and practices may be defined, the definitions are such that they cannot be operationalised. Since they cannot be operationalised, they cannot be tested or generalised.
3.2.2 Applied Science Conception

The conception of an applied science design discipline describes a practice in the form of 'specify and implement and test' and knowledge in the form of 'guidelines'. In the applied science conception of design, artifacts are still designed by a process of construction and evaluation and reconstruction. However, knowledge in the form of guidelines, derived from scientific knowledge, is used to guide the process. The conception of HCI as an applied science discipline is represented in Figure 3.7.

![Figure 3.7 Conception of HCI as an Applied Science Discipline](image)

By considering examples of HCI as an applied science discipline, Long and Dowell conclude that the knowledge and practices of HCI as applied science are derived from scientific theories that are operationalised, tested and generalised. However, the knowledge and practices themselves are not operationalised, tested and generalised with respect to the general problem of...
HCI. The Long and Dowell view of the relationship between scientific knowledge and applied science design can be related to the conceptions of scientific frameworks presented in Section 3.1. Figure 3.8 presents a view of the relationship between Kuhn's conception of science and Long and Dowell's conception of applied science.

Disorders of different types attempt to solve different general problems. Scientific disciplines attempt to solve the problem of understanding, whereas engineering disciplines attempt to solve the problem of design. Transforming discipline knowledge and practices that have been operationalised, tested and generalised with respect to the scientific problem of understanding, does not necessarily lead to knowledge that can be operationalised, tested or generalised with respect to the engineering problem of design.
3.2.3 Engineering Conception

The conception of an engineering design discipline describes a practice in the form of 'specify then implement' and knowledge in the form of 'principles'. In the engineering conception of design, artifacts are designed by a process of specification followed by a process of implementation. The process is supported by knowledge in the form of principles. The application of principles to the design process provides a guarantee that the artifact will satisfy the clients requirements. The conception of HCI as an engineering discipline is represented in Figure 3.9.

![Diagram](image)

Figure 3.9 Conception of HCI as an Engineering Discipline (following [Long & Dowell 89])

The knowledge and practices of an engineering discipline are defined, operationalised, testable and generalisable. Long & Dowell claim that, as of yet, there are no engineering principles in HCI. Their engineering conception is a conception for prescriptive HCI.
3.2.4 Conclusion

In Section 3.1 it was stated that Kuhn's notion of knowledge for use would be key in developing the conception of the framework for formal language design. From Long and Dowell's analysis presented in this section, it can be seen that scientific knowledge and practices and their derivatives are not necessarily the most useful form of knowledge for design. The next section will present an outline for the conception of the framework for formal language design by extending the Long and Dowell notion of engineering.
3.3 A Generic Engineering Conception.

This section presents a generic conception of a framework to support engineering design. It builds upon Kuhn's framework of a scientific discipline of Section 3.1 and the Long and Dowell engineering conception of Section 3.2. The generic conception is instantiated in Chapter 4, which presents a conception of a framework to support the engineering of formal languages.

This section contains the following sub-sections:

- Disciplinary Matrix and Scope of General Problem
- The Phenomena of Design
- Design Practice Exemplars
- Design Research Exemplars

The sub-sections consider the relationships between the components of Kuhn's framework of a scientific discipline and Long and Dowell's conception of an engineering discipline. Each sub-section presents requirements for a generic engineering conception. The first section discusses Kuhn's concept of the Disciplinary Matrix and the Long and Dowell notion of the Scope of the Discipline Problem. The second section considers how the scope of the discipline problem reflects the phenomena of the design problem. The third section considers Kuhn's notion of Shared Exemplar and considers the equivalent in design practice. The fourth section considers Kuhn's notion of Shared Exemplar and considers the equivalent in design research. The conclusion presents the generic engineering conception that results from the requirements presented in each of the previous sub-sections, and summarises how Kuhn's conception of a scientific discipline has been augmented in the light of Long and Dowell's work.
3.3.1 Disciplinary Matrix and Scope of General Problem

This section discusses Kuhn's concept of the Disciplinary Matrix and the Long and Dowell notion of the Scope of the Discipline Problem.

Section 3.1.4 introduced Kuhn's notion of a Disciplinary Matrix which consists of the symbolic generalisations, metaphysical assumptions and system of values of the discipline. Section 3.2 introduced Long and Dowell's concept of a Discipline Problem and its scope. By introducing the notions of desired performance and an interactive worksystem exhibiting actual performance, Long and Dowell are explicitly outlining some of the metaphysical assumptions and systems of values of the discipline of HCI. Thus, describing the scope of a discipline problem amounts to providing a component of a Disciplinary Matrix. The remaining component of a Disciplinary Matrix is discussed in conclusion at the end of this section.

Kuhn asserts that the value of knowledge is in its usefulness in solving problems. Design problems have two key components: the Requirements component and the Artifact component. The Requirements component represents the 'what' of the design problem. In Long and Dowell's terms, the Requirements component is the desired performance for an interactive worksystem. The Artifact component represents the 'how' of the design problem. In Long and Dowell's terms, the Artifact component is the interactive worksystem together with the actual performance it exhibits.

The conception of the framework for the design of formal languages must describe the scope of the general problem of formal language design. The description of the problem should describe the Requirements component and the Artifact component of the problem and the nature of the relationship between them.
3.3.2 The Phenomena of Design

Kuhn's framework is grounded in perceived phenomena. It is possible to distinguish two types of phenomena associated with design problems, the phenomena associated with the Requirements component and the phenomena associated with the Artifact component (Figure 3.10).

![Figure 3.10 The Phenomena of Design Problems](image)

In Figure 3.10, the phenomena associated with the Requirements component of the design problem are termed the Client Requirements. A Client here refers to an individual, or an organisation whose requirements may consist of the, possibly conflicting, requirements of sub-organisations and individuals. The phenomena associated with the Artifact component of the design problem are termed the Artifact. For any design problem it is necessary that there be techniques for determining whether Artifacts fulfil or satisfy Client Requirements. These techniques are termed empirical since they apply to phenomena. In Figure 3.10, the term Empirical Derivation is used to describe a technique for developing an Artifact from Client Requirements and the term Empirical Validation is used to describe a technique for determining if an Artifact satisfies Client Requirements.

The success of any design framework depends upon its ability to support the design of Artifacts which satisfy Client Requirements. The conception of the framework for the design of formal languages must have a set of empirical techniques for establishing the relationship between Artifact and Client Requirements.
3.3.3 Design Practice Exemplars

Lying between the Disciplinary Matrix and Phenomena in Kuhn's framework is the concept of the Paradigm of Shared Exemplars. For a scientific discipline, the general problem is one understanding by means of explanation and prediction. Thus Kuhn's shared exemplars form understanding as explanations and predictions. For an engineering discipline, the general problem is one of design, so one form of shared exemplars are design examples representing abstractions of Client Requirements and Artifacts and the relationships between these abstractions (Figure 3.11).

![Figure 3.11 Design Problems (Phenomena and Shared Exemplars).](image)

In Figure 3.11 abstraction of Client Requirements is termed the Specific Requirements Specification, whereas the abstraction of Artifact is called Specific Artifact Specification. The term 'Specific' is used, since these abstractions are specific to the particular design problem being considered. The relationships between the phenomena and abstractions, since they involve phenomena, are empirical. The relationship between the Specific Requirements Specification and the Specific Artifact Specification may be formal. The term 'formal' is employed in the sense of the definition of Chapter 2. Thus, it is assumed that the Specific Requirements Specification and the Specific Artifact Specification and the Derivation and Verification relationships between them are represented in some language that is formal with respect to a class of formal language designers and the task of formal language design.
The conception of the framework for the design of formal languages must have each of the following:

- A set of empirical techniques for establishing the relationship between Client Requirements and Specific Requirements Specifications.
- A set of empirical techniques for establishing the relationship between Artifact and Specific Artifact Specification.
- A set of formal techniques for establishing the relationship between Specific Requirements Specification and Specific Artifact Specification.

3.3.4 Design Research Exemplars

For a scientific discipline, the practice of the discipline is the research which aims to construct and validate knowledge which supports understanding in the form of explanation and prediction. For an engineering design discipline, whose knowledge is defined, operationalised, tested and generalised with respect to a general problem of design, there is a distinction between practice and research. Engineering practice involves employing engineering knowledge to solve specific design problems, whereas engineering research involves the construction and validation of engineering knowledge.

Thus, for an engineering discipline, there is an alternative but equivalent of Kuhn's shared exemplars that is examples of engineering research. Since engineering research constructs and validates knowledge which is defined, operationalised, tested and generalised with respect to the problem of design, engineering research exemplars, consist of abstractions of Specific Requirements Specifications and Specific Artifact Specifications and the relationships between them (Figure 3.12).
In Figure 3.12 abstraction of Specific Requirements Specification is termed the General Requirements Specification, whereas the abstraction of Specific Artifact Specification is called the General Artifact Specification. The term 'General' is used, since these abstractions are general to the particular classes of design problems. The relationships between all the abstractions are formal in the sense of Chapter 2.

The conception of the framework for the design of formal languages must have each of the following:

- A set of formal techniques for establishing the relationship between Specific Requirements Specifications and General Requirements Specifications.
- A set of formal techniques for establishing the relationship between General Requirements Specifications and General Artifact Specifications.
- A set of formal techniques for establishing the relationship between General Artifact Specifications and Specific Artifact Specifications.
3.3.4 Conclusion

This conclusion presents the generic engineering conception that results from the requirements presented in each of the previous sub-sections, and summarises how Kuhn's conception of a scientific discipline has been augmented in the light of Long and Dowell's work.

Summarising the requirements for the generic engineering conception outlined in the previous sections, an engineering conception should:

- Describe the scope of the general design problem in terms of Requirements and Artifact components and the relationship between them.
- Provide empirical techniques for empirically relating Client Requirements and Artifact at the level of the phenomena.
- Provide empirical techniques for relating Client Requirements and Artifact phenomena to Specific Requirements and Artifact Specifications.
- Provide formal techniques for relating Specific Requirements and Artifact Specifications.
- Provide formal techniques for relating Specific Requirements and Artifact Specifications to General Requirements and Artifact Specifications.
- Provide formal techniques for relating General Requirements and Artifact Specifications.

The generic engineering conception is illustrated graphically in Figure 3.13.
In this section, Long and Dowell's view of an engineering design discipline has been used to adapt Kuhn's framework of a scientific discipline to produce a generic conception of engineering design. Kuhn's approach has been adapted by adding:

- The validation of discipline knowledge with respect to design problems.
- The two forms of shared exemplar, engineering practice exemplars and engineering research exemplars.
- A General Design Problem with a clearly defined scope. The problem being expressed in terms of Requirements and Artifact.

![Diagram of the Generic Conception of Engineering Design]

Figure 3.13 The Generic Conception of Engineering Design.

The validation of the knowledge and consequent practices of engineering design occurs through the establishment of an empirical relationship between the Client Requirements and the Artifact. It is through this validation that engineering knowledge is generalised with respect to the problem of design. Such an approach to validating design knowledge is not unique. Carroll's use of pay-off evaluation and its relationship with design rationale [Carroll et al 92] provides another example.
The two sorts of shared exemplar distinguish clearly between the activities of engineering practice and engineering research. Engineering practice is concerned with solving specific design problems using general knowledge. Engineering research is concerned with constructing and validating general knowledge. The formal aspects of the generic engineering framework relate the specific abstractions associated with specific design problems to the general abstractions associated with the knowledge that supports design. It is precisely these formal aspects of the framework that form the symbolic generalisation component that was missing from the discussion of the disciplinary matrix in Section 3.3.1.

This section has presented a generic conception of engineering design based upon the work of Kuhn and Long and Dowell. In Chapter 4, this generic conception is instantiated for the case of formal language design by describing the scope of the formal language design problem. The next section exploits the generic engineering framework to resolve some framework related issues that have arisen in the field of software engineering, thus demonstrating the potential utility of the framework.
3.4 Formality and Design

This section considers the role of formality in software engineering. It discusses framework related disagreements that have occurred concerning the application of formal methods to software design, in what one author has termed the 'Formal Design Paradigm' [Dasgupta 91]. The thesis is concerned primarily with the formal aspects of a framework for formal language design. A generic conception of a framework to support design was established in the previous section. The discussions of the perceived problems of formal methods in software engineering and their resolution in the light of the generic engineering framework illustrate the potential utility of the framework. Two aspects of formal methods for software engineering are considered; program verification and executable specification.

3.4.1 Program Verification

"Computer programming is an exact science, in that all properties of a program and all consequences of executing it can, in principle, be found out from the text of the program itself by means of purely deductive reasoning." [Hoare 69]

The above quote, repeated from Chapter 1, is from a paper which was instrumental in founding the field of Program Verification, which can in turn be thought of as forming part of the Formal Design Paradigm.

A naive conception of the process of program verification is illustrated in the Figure 3.14:

![Figure 3.14 A Naive Conception of Program Verification.](image-url)
A Specification describes some required program at a level of detail understandable by a human being. In other words, a specification describes what the program should do. A Program describes, in detail understandable to a computing machine, how the specification is to be satisfied. A program is verified within some formal system if it can be demonstrated that the program will achieve the required transformation described by the specification.

The problem with the above description, and the quote from Hoare, is that the distinction between the world of formality and reason and the empirical phenomena is blurred. A slightly more sophisticated conception of program verification is presented in the Figure 3.15:

![Figure 3.15 A Revised Conception of Program Verification.](image)

In this conception of verification, the Specification is empirically derived from the Client Requirements. The program is formally verified against the Specification. Provided the Specification is a true reflection of the Client Requirements, a formally verified Program is certain to satisfy the Client Requirements. Even behind this weakened view of program lies a strongly rationalist-absolutist position concerning the nature of knowledge. It is this underlying position that has been the cause of much recent controversy.

The philosopher Fetzer, citing philosophers of science, has claimed that the very idea of program verification as expounded above is impossible [Fetzer 88]. Fetzer's objections are based upon the failure of the Program Verification community to draw a distinction between the notion of an Algorithm, a formal abstraction and a Program, an empirical phenomena which is the
reflection of the Algorithm. Algorithms can be formally verified against Specifications, as in the Figure 3.16:

![Figure 3.16 Formal Verification of an Algorithm](image)

But programs can only be empirically validated as in the Figure 3.17.

![Figure 3.17 Empirical validation of a program](image)

In fact, Fetzer takes a falsificationist position and claims that programs can only be refuted. Fetzer believes further that the distinction between Client Requirements and Specification and Algorithm and Program are too great to make program verification feasible as part of the design process.

As can be imagined, Fetzer's article prompted angry responses from the program verification community. The debate is summarised by another philosopher Barwise [Barwise 89].
3.4.2 Executable Specification

Another, less well-known issue that has arisen, concerning the Formal Design Paradigm involves the concept of an Executable Specification [Turner 85]. Basically, an Executable Specification is a specification derived from Client Requirements that can be directly executed. This situation is represented in the Figure 3.18:

![Figure 3.18 Executable Specification](image)

The aim of Executable Specification is to provide a means of validating the Specification against the Client Requirements. Often the executable specification might be constructed only for some particularly important aspect of the Client Requirements. The executable specification is open to empirical validation in the same sense as a Program.

Two well-respected researchers, working within the formal design paradigm have objected to the notion of executable specification. They seek:

"to warn of the dangers of limiting specification languages to the point where all their constructs can be executed." [Hayes & Jones 89]

Essentially, the authors claim that unnecessary constraints are placed upon specification languages, if their constructs are to be executed. They draw a distinction between a Specification and a Prototype, which can serve the purpose of validating requirements. This distinction can be represented as in Figure 3.19:
In this view, Prototypes are (usually) partial implementations which can be formally verified against the Specification. The idea is that Prototypes can be developed rapidly from the Specification. It is interesting to note that Hayes and Jones although recognising the need for the distinction between Specification and Prototype, fail to make a distinction between the Prototype Algorithm and Prototype Program, required by Fetzer in the previous section.
3.4.3 Conclusion

The generic engineering conception (Section 3.3) can be used to shed light on the conceptual issues of Program Verification and Executable Specifications. Figure 3.20 is the part of the Figure 3.13, representing the generic engineering conception. Figure 3.20 consists of the Requirements and Artifact components of the general design problem at the level of phenomena and specific abstractions.

![Diagram](image)

Figure 3.20 The Generic Engineering Conception: Phenomena and Specific Abstractions.

The generic engineering conception can be related to the discipline of Software Engineering according to the following table:

<table>
<thead>
<tr>
<th>Generic Engineering Concept</th>
<th>Software Engineering Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Requirements</td>
<td>Client Requirements</td>
</tr>
<tr>
<td>Specific Requirements Specification</td>
<td>Specification</td>
</tr>
<tr>
<td>Specific Artifact Specification</td>
<td>Algorithm</td>
</tr>
<tr>
<td>Artifact</td>
<td>Program</td>
</tr>
</tbody>
</table>

Fetzer's objections to the concept of Program Verification can now be seen to be based upon the failure of the Program Verification community to distinguish between Specific Artifact Specification and Artifact.
Hayes and Jones's objections to Executable Specification can be seen to be due to the failure of workers in the field to distinguish between the Specific Requirements Specification and Specific Artifact Specification.

It can be seen from the two examples above that the generic engineering conception is capable of resolving some of the problems encountered in the 'Formal Design Paradigm'.

By considering the generic engineering conception further, it is possible to consider an aspect of formal work, not often considered in the field of Software Engineering - the notion of formal knowledge.

The General Requirements and Artefact Specifications and the relationships between these specifications that form the formal aspects of the knowledge of generic engineering conception do not seem to have a ready analogue in the Formal Design Paradigm. Two options present themselves concerning the nature of such an analogue.

General formal knowledge might be embedded in the formal system which establishes the formal relationship between the Algorithm and Specification. This option is a strongly rationalist-absolutist one, equating knowledge with reason.

A second option would relate general formal knowledge to the notion of reusable software components. Thus, knowledge would consist of some form of General Specification and a General Algorithm which corresponds to some already implemented General Program. Knowledge would be applied by instantiating the General Specification to some Specific Specification and the General Algorithm to a Specific Algorithm. An example of the second form of general formal knowledge would be the Specification and Algorithm for a generic list which could be instantiated to form a list of numbers, employee records, etc.

The second notion of knowledge, linking formality to reuse, may go a long way to countering Fetzer's objections that formality is too costly to be of use in the design of computer systems.
In this section, it has been seen that the generic engineering conception can shed light upon problematic issues concerning formality and design. These issues of formality also justify the distinction between the formal and empirical relationships of the conception.

Following Fetzer's arguments, both Client Requirements and Artifact are empirical phenomena and can only be related to each other and to Specific Requirements and Artifact Specifications using empirical techniques. Specific Requirements and Artifact Specifications and General Requirements and Artifact Specifications are abstractions in some formal language and can thus be related using formal techniques.
3.5 Conclusion

This chapter has considered perspectives from the philosophy of science, including those of Lakatos, Kuhn and Feyerabend. Kuhn's framework, because of its notion of knowledge as a means of solving problems, was chosen as the basis for developing the framework for the design of formal languages. It was pointed out that the framework, in line with Feyerabend's perspective, was intended to support rather than constrain the activities of formal language designers.

Kuhn's framework was extended in line with the work of Long and Dowell, which contrasts the nature of the general problem of science with that of the general problem of design. The result of this extension was the generic conception of engineering design presented in Section 3.3. Chapter 4 instantiates this generic conception to a conception of formal language design by describing the scope of the general problem of language design.

The final section demonstrated the utility of the generic engineering conception in other fields of design, by indicating how it can be used to address the problematic issues concerning formality and design in software engineering. This demonstration also acted to justify the distinction between the formal and empirical techniques, as they are used in design.

Chapter 4 describes the scope of the general design problem and thus explicitly outlines the metaphysical assumptions and system of values of Kuhn's disciplinary matrix. The remainder of the thesis outlines the formal aspects of the framework, and thus explicitly outlines the symbolic generalisations of Kuhn's disciplinary matrix.
4 An Engineering Conception of Formal Language Design

This chapter instantiates the generic conception of an engineering discipline developed in Chapter 3 for the purpose of developing a conception of formal language design. This conception is then instantiated further with the development of a formal framework in Chapters 6 to 10.

The first section employs the definition of formal language that was established in Chapter 2 to outline the scope of the general formal language design problem. The second section considers the requirements aspects of the scope of the general design problem in more detail. The third section considers the artifact aspects of the scope of the general design problem in more detail. The fourth section considers how the work of other disciplines relates to the conception of formal language design. The fifth section considers the relationship between the conception outlined here and the Long and Dowell conception of HCI [Long & Dowell 89], which was used to construct the generic engineering conception in Chapter 3.
4.1 Outline of the Scope of the General Design Problem

This section employs the definition of formal language that was established in Chapter 2 to outline the scope of the general formal language design problem.

Recall the definition of formal language given in Chapter 2:

A language is *formal* with respect to a class of agents, and a task, if it can be interpreted and employed by those agents so as to carry out the task.

Recall also from Chapter 3:

The conception of the framework for the design of formal languages must describe the scope of the general problem of formal language design. The description of the problem should describe the Requirements component and the Artifact component of the problem and the nature of the relationship between them.

The Artifact component in formal language design is considered to be the languages together with the agents that employ the languages. The Requirements component is therefore composed of some description of the task. A task may be thought of as of having two components: benefit and cost. The work performed by the agents who interpret and employ the languages brings about work benefits. Performing the work also incurs costs.

The name given to an Artifact of formal language design is a formal language worksystem. A *formal language worksystem* is composed of formal languages, and language agents who interpret and employ the formal language to achieve the work benefits of the worksystem, while incurring the worksystem costs. An example worksystem is presented in Figure 4.1.
The worksystem depicted in Figure 4.1 consists of two agents Agent 1 and Agent 2 which employ Formal Language 1 to achieve desired benefits while incurring acceptable costs.

The Requirements component of a formal language worksystem is composed of work benefits and worksystem costs. Worksystems perform work by transforming work structures. The work benefit of a worksystem is considered as a transformation of work structures. Different work benefits derive from different transformations of structures.

In transforming work structures, worksystems incur worksystem costs. Different worksystems will typically bring about different transformations of structure while incurring different worksystem costs.
Thus, different worksystems will typically produce different work benefits, while incurring different worksystem costs. The combination of the work benefits produced by a worksystem, together with the costs incurred by the worksystem, is called the worksystem performance.

Specifying the requirements for a formal language worksystem involves specifying a collection of acceptable worksystem performances. Thus, a formal language worksystem will satisfy the requirements, if it achieves the work benefits and worksystem costs of some acceptable performance.

This section has outlined the scope of the general problem of formal language design. The next section considers the requirements component of the general problem in more detail.
4.2 Requirements: Worksystem Performance

This section considers the requirements aspects of the scope of the general design problem in more detail.

In Section 4.1, it was stated that specifying the requirements for a formal language worksystem involves specifying a collection of acceptable performances. It is not enough, however, for requirements specification only to specify such a collection. Consider, for example, the situation in which two worksystems both achieve some acceptable performance, but one is clearly superior, in terms of both work benefits and worksystem costs. In such a situation, the requirements specification should enable a worksystem designer to decide which system to implement. Thus, in addition to specifying a collection of acceptable performances, it is also necessary to specify an ordering of the performances.

The ordering of performances is dependant on an ordering of work benefits and an ordering of worksystem costs. The work structure transformations, that are the interest of the client, who requires a formal language worksystem is called the work domain. The benefit ordering represents the ordering of the work domain according to client preferred benefits. The desired benefits of a formal language worksystem are the collection of elements of the work domain, whose benefits are desired by the client. The work domain, the benefit ordering and the desired benefits of a worksystem are together called the benefit requirements of the system.

The cost ordering represents the ordering of possible work costs according to the client preferred costs. The desired cost of a formal language worksystem is the collection of desired worksystem costs of the system. The cost ordering and the desired costs of the worksystem are together called the cost requirements of the worksystem.

A performance ordering is a combination of benefit orderings and cost orderings. The desired performance of a formal language worksystem is a combination of the desired benefits with the desired costs. The performance ordering and the desired performance of the worksystem are together called the performance requirements of the worksystem.
Thus the Performance Requirements for a formal language worksystem can be expressed by outlining:

- The performance ordering
- The set of desired performances

The Performance Ordering is a combination of:

- The benefits ordering
- The costs ordering

The set of desired performances is a combination of:

- The desired benefits
- The desired costs

Work benefits are expressed in terms of work structure transformations.

This section has outlined in detail the requirements component of the general formal language design problem. The next section considers the artifact aspects of the problem.
4.3 Artifact: Language Worksystems

This section considers the artifact aspects of the scope of the general design problem in more detail. Having completed the discussion of requirements in the previous section, it is possible to further develop the notion of a formal language worksystem outlined in Section 4.1. This development is achieved by elaborating upon the concept of language agent and detailing more precisely what is meant by interpreting and employing languages.

In general, language agents may be of any form capable of understanding and employing language for the purpose of transforming work structures. Thus, language agents may be humans, computers or even other formal language worksystems.

An agent can be said to have interpreted a structure of a language, if the perception of the language structure results in some transformation of work or some further employment of language structures. An agent can be said to have employed a structure of a language, if that structure has been developed in response to some perception of work structures or other language structures.

The requirements specification for individual agents may be expressed in the same manner as the requirements specifications for the language worksystem as a whole. In addition, the requirements for individual agents are related to the requirements of the system as a whole. Thus, the requirements for Agent 1 of Figure 4.1 might take the form of Figure 4.3.

In Figure 4.3., the work structures of Agent 1 consist of some of the work structures of the system as a whole, together with some of the structures of Formal Language 1.
Part of the Work of whole worksystem

Formal Language 1

Agent 1

Work Benefit relates to work benefit of whole worksystem

Agent 1 costs are a component of the costs of the worksystem as a whole

Figure 4.3 The Form of the Requirements Specification of Agent 1 of Figure 4.1.

Thus, the specification of a formal language worksystem consists of:

- A set of formal language specifications
- A set of language agent requirement specifications

A formal language worksystem achieves a desired performance if, when all the language agent requirement specifications are achieved, the worksystem achieves desired work benefits, at desired worksystem costs.

This section has outlined in detail the artifact component of the general formal language design problem. The next section considers disciplines that are related to formal language engineering and the nature of these relationships.
4.4 Related Disciplines

This section considers how the work of other disciplines relates to the conception of formal language design. Disciplines can relate to the conception of the design of formal languages according to their own conception. They may relate in one of two ways. They may either have a component conception or have an intersecting conception.

If a discipline has a component conception, that discipline could help to solve the formal language design problem. With the conception of formal language engineering, disciplines with component conceptions arise with language agents. According to the conception, artifact specifications specify the requirements of language agents. To specify the artifact that corresponds to a language agent is the job of the discipline with the component conception. The particular discipline is dependant upon the language agent.

For example, a requirement of an agent might involve employing a formal language by transforming natural language queries concerning some database to the structured query language of the database management system. The selection and training of the human being to carry out this operation is the task of the discipline of Human Factors.

As part of the same formal language system, a requirement of an agent might be the transformation of a query in the structured query language to a report outlining all the entities that satisfy the query. The selection of a database management system and computing machine and implementation of a database to support the queries is the task of the discipline of Software Engineering.

The second form of related discipline is a discipline with an intersecting conception. Disciplines with intersecting conceptions, at least in part, attempt to solve the same problem. Such disciplines arise because different disciplines have different conceptions whose spheres of interest overlap.

ICI forms an example of a discipline whose conception intersects with formal language engineering conception. In situations where a design is postulated involving formal languages, with human and computer agents,
then an intersection occurs between HCI and formal language design. HCI is an obvious example, since much of the work here is based upon work in that field [Dowell & Long 89]. Indeed, in the next section of this chapter the conception of formal language engineering is compared to the Dowell and Long conception of HCI.

Clearly there are other intersecting activities to formal language engineering. When worksystems involve a group of human agents, one might consider the field of Computer Supported Co-operative Work (CSCW) [Hughes, Randall & Shapiro 91] to be related. The design of methods to aid in the development of computer systems, whether these are the traditionally termed 'Formal Methods' such as Z [Spivey 89] and VDM [Jones 86] or structured methods such as Yourdon [Yourdon 89] or JSD [Jackson 83] is also related. The similarity of some approaches to method and case tool development [Nuseibeh & Finkelstein 92] provide further evidence of such relationships.

This section has considered the relationship between formal language engineering and other engineering activities. The next section considers the relationship between formal language engineering and the Long and Dowell conception of HCI engineering.
4.5 Language Engineering and HCI Engineering

This conception of formal language design is based closely upon the engineering conception of HCI (HCIE) outlined in Long and Dowell, [Long & Dowell 89] and further expanded in Dowell and Long [Dowell & Long 89]. This section considers the differences with the conception of formal language engineering.

The formal language and HCIE conceptions both assert a fundamental distinction between work (the requirements) and worksystem (the artifact). At the level of requirements specification, the conceptions are much the same, although the terminology is different. These differences can be summarised in the following table:

<table>
<thead>
<tr>
<th>HCI Engineering</th>
<th>Formal Language Engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain of Application</td>
<td>Work Domain</td>
</tr>
<tr>
<td>Work = Transformation of Objects</td>
<td>Work = transformation of work structures</td>
</tr>
<tr>
<td>Quality</td>
<td>Benefit</td>
</tr>
<tr>
<td>Desired Quality</td>
<td>Desired Benefit</td>
</tr>
<tr>
<td>Costs</td>
<td>Costs</td>
</tr>
</tbody>
</table>

As the table indicates, there are two key differences in terminology. The first is minor and concerns the use of the word benefit instead of quality, based upon the notion that benefit provides a better contrast with cost.

The second distinction in terminology concerns the HCIE concept of a domain object which relates to a formal language concept of a work structure. HCIE uses the concept of structure to describe aspects of HCI worksystems. In the formal language conception, the concept of structure is used in a much wider sense.

There is a further distinction that can be drawn at the level of requirements specification and that is between the different notions of desired performance. The formal language conception elaborates upon the notions of performance of HCIE. The notion of performance orderings is part of this elaboration. The
consideration of desired performance as a set, rather than as an individual performance, constitutes the other part of the elaboration.

At the level of the artifact specification, the distinction between the HCIE and formal language conception grows stronger. For HCIE, a worksystem consists of humans and computers. Both humans and computers exhibit structures and behaviours. The behaviours interact to achieve some actual performance of work.

Thus, for the HCIE conception, an artifact specification is a specification of computer and human structures and behaviours. Whereas, for the formal language conception the artifact specification consists of formal language specifications and language agent requirement specifications.

There are a number of consequences of this distinction. Firstly, the notion of structures and behaviours captures the dynamic nature of human computer interaction in a much better fashion than the notion of language. Secondly, the clear distinction between the objects of the domain and the structures and behaviours of the worksystem highlights the difference between the requirements specification and the artifact specification.

On the other hand, it can be argued that the formal language conception has traded the sophisticated view offered by structures and behaviours for a simpler view. In this view, the work of the worksystem is the transformation of work structures. The work of the language agents, the components of worksystems, is the transformation of work and language structures. Thus, in the formal language conception, there are only structures and transformations. This simplification will make it easier to develop a framework, although the framework may as a result be less effective in supporting the design of human-computer worksystems.

It can also be argued that the HCIE conception over-emphasises the distinction between requirements and artifact specifications, failing to recognise that one discipline's requirement specification is another discipline's artifact specification. Thus, for the HCIE conception there is no distinction between disciplines with component conceptions, which solve part of the problem, disciplines with intersecting conceptions, which solve the same problem. Hence, the HCI conception, defines the common ground
between the sub-disciplines of Human Factors and Software Engineering rather than a separate discipline.

The HCIE and Formal Language conceptions are both instantiations of the generic engineering conception of design. They are distinct in terms of scope and focus. The HCIE conception has a specific scope and a sharp focus on the structure and behaviour of human computer worksystems. The formal language conception has a much broader scope, but a softer focus resulting in a more simplistic view of worksystems.
4.6 Conclusion

Chapter 2 developed a consideration of the nature of formality which resulted in the definition of the notion of formal language. Chapter 3 developed a consideration of the nature of frameworks that support disciplines of both science and design, and resulted in the definition of a generic engineering design conception. This chapter has used the definition of formal language developed in Chapter 2 to describe the scope of the general formal language design problem, and, in doing so, has instantiated the generic engineering conception for the case of formal language engineering.

Section 1.7 outlines the criteria for assessing this thesis. The criteria that are applicable to Chapters 2, 3 and 4 are concerned with the level of external synthesis at the level of the thesis. Recall from Section 1.7 that assessing the external synthesis at the level of the conception requires an answer to the question:

How useful is the conception as an approach to the design of complex systems?

Chapter 2 developed a synthesis of different notions of formality, derived from philosophers of mathematics, as well as those concerned more directly with formal language, or, as some call it, notation. The following definition of the concept of formal language was given:

A language is formal with respect to a class of agents, and a task, if it can be interpreted and employed by those agents so as to carry out the task.

With this definition the concept of a formal language can be seen as arising in all complex systems which consist of agents that communicate in order to fulfil some purpose.

Chapter 3 developed a synthesis of different notions of framework, derived from philosophers of sciences, as well as those concerned more directly with the nature of design knowledge. This synthesis resulted in the description of a generic engineering framework oriented specifically towards design. It was
further demonstrated that this generic conception has the potential to resolve many of the issues that arise with the use of formality in software engineering.

This chapter has developed a synthesis of the work of Chapters 2 and 3, resulting in the conception for formal language design. It is therefore argued that the conception outlines a promising approach to the design of complex systems. Whether this promise is fulfilled will depend upon the nature of the framework that forms an instantiation of the conception. The remaining chapters of this thesis outline this framework. The next chapter outlines the notations that will form the basis of the description of the framework.
5 A Basis for the Formal Framework

The aim of this chapter is to outline the basis for a formal framework for language design. The scope of the formal framework is illustrated in the Figure 5.1.

Within the framework, it must be possible to describe all the formal derivations and validations. In order to achieve these descriptions it is necessary to describe the following components:

- Specific Requirements Specification
- Specific Artifact Specification
- Engineering Principles

The Specific Requirements Specification is to be expressed in the terms of the Desired Performance and Performance Ordering relationships. The Specific Artifact Specification has to describe Formal Languages and the Desired Performance and Performance Ordering of Language Agents. The
Engineering Principles reflect the form of Requirements Specification and Artifact Specification at a more abstract level.

The first section of this chapter presents a rationale for the basis of the framework, that is, Z extended with concepts from category theory. The second section introduces the Z notation. The third section introduces the concept of a category. The remaining sections introduce concepts from category theory that are employed in the rest of the thesis.
5.1 Choice of a Formal Framework

Recalling the definition of formal language given in Chapter 2:

A language is formal with respect to a class of agents, and a task, if it can be interpreted and employed by those agents so as to carry out the task.

What is being attempted here is the (informal) development of a formal language for formal language engineering. The language agents in this case are formal language designers.

There are two general options for the basis of the formal framework for language design. The framework may be purpose-built or based upon one or more existing formalisms. The purpose-built option has not been considered, since the amount of work required to construct such a basis would not be possible given the time period of the PhD.

Considering existing formalisms, there is a broad spectrum of options for the basis for the formal framework for language design. At one end of the spectrum are general purpose formalisms, and at the other end are formalisms designed for more specific purposes. Figure 5.2 presents some of the formalisms and their position within the spectrum. Clearly, there could be much debate about exactly where to place different formalisms on such a spectrum. Figure 5.2 is meant only as a rough guide.

The formalisms being considered are generally what would be considered to be formal languages by the formal methods community. Notations such as QOC design rationale [MacLean et al 91] have not been considered. The reason for this limited consideration is to attempt to appeal to language designers who are already using formal notations.

Predicate logic [Hamilton 88] could be used as a basis for constructing a formal framework. Indeed, a predicate logic of some form is part of the language of most of the formalisms considered below. The advantages are that it is relatively easily learned and can be applied generally. Its
disadvantages arise, from its being such a small formalism. It therefore requires significant work, in terms of defining constants, functions and axioms, to create the constructs required for the framework. The work would almost be akin to developing a purpose-built framework. What is more, there is no means for structuring definitions. Predicate logic is therefore rejected as a sole basis for the framework for formal language design.

Constructive type theory [Martin-Löf 84a] has been advocated as a framework for computer science [Martin-Löf 84b], [Backhouse 90], [Reeves 88]. In terms used here requirements specifications can be considered as types and artifact specifications as instances of types. Constructive type theory is in line with the fundamental philosophy behind the thesis. The notation is larger than predicate logic, and it is possible to construct sophisticated objects more easily. However, its structuring mechanisms are poor, the notation is difficult to learn, and a relatively small number of people are familiar with it. Thus, constructive type theory is therefore rejected as a sole basis for the framework for formal language design.

Set Theory [Hamilton 88] is considered the usual foundations for mathematics. As with predicate logic, some form of set theory is a part of many of the formalisms considered below. As with predicate logic, its advantages are that it is relatively easily learned and can be generally applied.
Although set theory is a theory of predicate logic and is considerably richer than predicate logic on its own, it suffers from many of the same disadvantages. It still requires significant work to create the constructs and also lacks a sophisticated structuring mechanism. Set theory is therefore rejected as a sole basis for the framework for formal language design.

Mathematical category theory [Arbib & Manes 75] [Goldblatt 84], [MacLane 71] [ Rydeheard & Burstall 1988] has also been considered as a possible foundation for mathematics. Domain theory and the denotational approach to semantics, discussed below, can be easily embedded within it [Tennent 91]. It has also been used to generalise text-based production rules, to production rules for graphs [Ehrig et al 91]. Category theory could also be used to consider the higher level issues such as formal derivation and validation relationships. Such an approach has been suggested for planning [Ho 82]. Thus, of all the relatively general formalisms considered so far, category theory appears to be the most promising. However, as with all the others, it lacks the sophisticated structuring mechanisms of some of the formalisms considered below. Thus, category theory is rejected as a sole basis for the framework for formal language design.

Production Rule formalisms have long been associated with language research [Hopcroft & Ullman 79]. They have been used to describe the syntax of text-based and other languages [Gips & Stiny 80]. Some notion of the production rule will be of central importance in the framework. However, production rules by themselves do not possess significant structuring mechanisms. Thus, production rules are rejected as a sole basis for the framework for formal language design.

The Domain Theory of Scott and Strachey [Stoy 77] may also be considered as a basis for a framework. Domain theory forms the basis of the denotational approach to defining the semantics of formal languages. The domains considered are not work domains as defined in Chapter 4, but abstract mathematical domains. A revised notion of denotational semantics will be of relevance to formal language specification. However, domain theory is too specialised to be of general use in forming a basis for the framework for language design.
The Z Schema notation [Spivey 89] is essentially an enrichment of set theory to provide the missing structuring mechanisms. It was developed for constructing artifact specifications. It has been used to describe computer systems as a whole [Hayes 87] and the user interface in particular [Sufrin 86]. The Z notation therefore seems very attractive as the basis for the formal framework.

The Z notation is an example of what has been termed a model-based specification technique [Sommerville 89]. The other major model-based notation is that of the Vienna Development Method (VDM) [Jones 86]. As with Z, VDM has been used for artifact specification. In contrast to Z, however, VDM is oriented much more to the specification of computer systems. The VDM notation is not so general purpose. Thus, Z is preferred to VDM as a basis for the formal framework for language design.

Another range of formalisms that are on a par with VDM are the algebraic languages such as OBJ [Futatsugi et al. 85] and Larch [Guttagef et al. 85]. As with VDM, these algebraic languages are much more oriented towards the specification of computer systems in terms of abstract data types. Thus, Z is preferred to the algebraic specification techniques as a basis for the formal framework for language design.

Process Algebras such as CSP [Hoare 85] and CCS [Milner 89] have been used to describe the behaviour of communicating agents. They have also been advocated as a formalism suitable for HCI [Alexander 87], [Stork 92]. These formalisms can deal with notions of communication, concurrency and nondeterminism with an algebraic representation of essentially state transition diagrams. Their emphasis, however, is on the communicating agent rather than on the language. Thus, language of communication is merely an alphabet of text-based symbols. These formalisms provide a valuable insight into the specification of agents, but are not capable of forming the sole basis for the framework of formal language design.

Reviewing the options above, two stand out. From the most general purpose group, category theory has been used to generalise notions of syntax and as a basis for defining the semantics of languages. From the specification languages, Z would appear to be the best, being significantly general purpose
and maintaining good structuring mechanisms. The framework therefore is based upon the Z notation extended with category theory to allow for the description of syntax and semantics. In the following sections of this chapter, the Z notation is introduced and extended with definitions from category theory.
5.2 The Z Notation

This section provides a brief introduction to the Z notation. Z provides two mechanisms to structure predicate logic with set theory. These mechanisms are Schemas and Axiomatic descriptions.

Schemas are considered first. Schemas are used to describe classes of objects. The objects have a number of attributes, each of a particular type. A simple schema has the following form.

```
SchemaName

<table>
<thead>
<tr>
<th>identifier1 : Type1</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifierN : TypeN</td>
</tr>
</tbody>
</table>
```

The identifiers represent the name of the attributes, the types represent the types of the attribute and the predicates represent the relations between attributes. In what follows, attributes are sometimes referred to as components.

An example schema is now presented which defines a class of dates. The schema has three attributes: day, a number from 1 to 31; month, a number from 1 to 12; and year, a natural number. A natural number is any whole number including zero. The set of all natural numbers is denoted in Z by the symbol \( \mathbb{N} \).
The schema contains three conditions in the form of predicates. They state:

- If the month is 4, 6, 9 or 11, then the day is less than or equal to 30. This condition ensures that for any month with 30 days the value of the day attribute will be 30 or less.
- If the month is 2, the day is less than or equal to 29. This condition ensures that for the month of February, the value of the day attribute will be 29 or less.
- If the month is 2 and the year modulo 4 is not zero, then the day is less than or equal to 28. This condition ensures that for the month of February that is not in a leap year, the value of the day attribute will be 28 or less.

Note that for any natural numbers, n, m, with m not zero, n modulo m, denoted in Z,

\[ n \mod m \]

is the remainder left when n is divided by m.
The schema date defines a class of tuples together with three functions for extracting the component values of the tuples. The tuples:

(1,1,1993)  
(2,10,54)  
(3,6,2001)

are all examples of tuples belonging to Date. The following tuples, although of the right type do not belong to date:

(31,4,1993)  
(30,2,1993)  
(29,2,1993)

The first tuple fails the first condition of the schema; the second tuple fails the second condition of the schema; and the third tuple fails the final condition of the schema.

The three functions for extracting the components of the tuple are the attribute identifiers. They can be applied as follows:

(1,3,1993).day = 1  
(1,3,1993).month = 3  
(1,3,1993).year = 1993

It is possible to construct schemas that are extensions of other schemas. The following example of a schema, which defines a class of date and times, extends the date schema. It consists of the name of the date schema, together with two additional attributes, hour and minute. It has no conditions.

```
<table>
<thead>
<tr>
<th>DateAndTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
</tr>
<tr>
<td>hour        : 1..24</td>
</tr>
<tr>
<td>minute      : 1..60</td>
</tr>
<tr>
<td>-------------</td>
</tr>
</tbody>
</table>
```
The previous schema is equivalent to writing the following schema. As can be seen, the schema extension notation is a useful device when constructing large classes:

```plaintext
DateAndTime

<table>
<thead>
<tr>
<th>Field</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>day</td>
<td>1..31</td>
</tr>
<tr>
<td>month</td>
<td>1..12</td>
</tr>
<tr>
<td>year</td>
<td>( \mathbb{N} )</td>
</tr>
<tr>
<td>hour</td>
<td>1..24</td>
</tr>
<tr>
<td>minute</td>
<td>1..60</td>
</tr>
</tbody>
</table>

month \( \in \{4,6,9,11\} \Rightarrow day \leq 30 \)

month = 2 \Rightarrow day \leq 29

(month = 2 \land (year \mod 4 \neq 0)) \Rightarrow day \leq 28
```
The classes defined by schemas can be used as the type for some attribute in another schema. In this manner, it is possible to have attributes that emerge at different levels in a hierarchy and to relate attributes across and within the hierarchy. The following schema defines a class of messages. It has three attributes. The attributes sent and received are both dates and the attribute message contents is a sequence of natural numbers.

```
Message

    sent :Date
    messageContents :seq N
    received :Date
```

\[\text{sent.year} \leq \text{received.year}\]

\[\text{sent.year} = \text{received.year} \Rightarrow \text{sent.month} \leq \text{received.month}\]

\[\begin{align*}
    \text{sent.year} = \text{received.year} & \land \text{sent.month} = \text{received.month} \\
    \Rightarrow \\
    \text{sent.day} & \leq \text{received.day}
\end{align*}\]

The conditions state that the date the message was sent is less than or equal to the date the message was received. The three conditions are as follows:

- The year in which the message is sent, is less than or equal to the year in which the message is received.
- If the year in which the message is sent is equal to the year in which the message is received, then the month in which the message is sent is less than or equal to the month in which the message is received.
- If the year and month in which the message is sent is equal to the year and month in which the message is received, then the day on which the message is sent is less than or equal to the day on which the message is received.
The message schema defined above has as its messages contents, sequences of natural numbers; but messages in general may have very different contents. The generic schema construct enables the definition of schemas over generic classes. The generic message schema that follows is identical to the original schema, except for the generic class of Contents:

```
Message [Contents]

sent : Date
messageContents : Contents
received : Date
```

\[
\begin{align*}
\text{sent.year} & \leq \text{received.year} \\
\text{sent.year} = \text{received.year} \Rightarrow \text{sent.month} & \leq \text{received.month} \\
(\text{sent.year} = \text{received.year} \land \text{sent.month} = \text{received.month}) & \Rightarrow \\
\text{sent.day} & \leq \text{received.day}
\end{align*}
\]
The conditions on both the generic and simple message schema are rather unwieldy. They would be much easier to state if there were an order relationship upon dates. The second form of structuring mechanism of the Z notation, the axiomatic description, provides a means for specifying such a relation. The simple form of the axiomatic description is as follows:

```
identifier1 :Type1

identifierN :TypeN

predicate1

predicateM
```

The axiomatic description introduces a number of global constants. The constants are denoted by the identifiers. The constants are of the type denoted by the corresponding type and obey the predicates.
The simple axiomatic description can be used to introduce a global ordering on dates. The ordering is a relation, denoted by the symbol $\leftrightarrow$, between dates. It is in infix form, indicated by the underscores either side, that is, it can be used in-between the two objects it relates.

\[
\leq \quad : \text{Dates} \leftrightarrow \text{Dates}
\]

\[\forall \ d, d' : \text{Dates} \quad d \leq d' \quad \iff \]
\[
\begin{align*}
& \quad \text{(d.year } \leq \text{d'.year} \\
& \land \quad \text{(d.year } = \text{d'.year } \Rightarrow \text{d.month } \leq \text{d'.month}) \\
& \land \quad \text{((d.year } = \text{d'.year } \land \text{d.month } = \text{d'.month}) \Rightarrow \text{d.day } \leq \text{d'.day})
\end{align*}
\]

The condition asserts that any date is less than another date exactly when:

- The year of the first date is less than or equal to the year of the second date.
- If the year of the first date is equal to the year of the second date, then the month of the first date is less than or equal to the month of the second date.
- If the year and month of the first date is equal to the year and month of the second date, then the day of the first date is less than or equal to the day of the second date.
The message can now be defined more simply using the date ordering:

<table>
<thead>
<tr>
<th>Message</th>
<th>[Contents]</th>
</tr>
</thead>
<tbody>
<tr>
<td>sent</td>
<td>: Date</td>
</tr>
<tr>
<td>messageContents</td>
<td>: Contents</td>
</tr>
<tr>
<td>received</td>
<td>: Date</td>
</tr>
</tbody>
</table>

\[ \text{sent} \leq \text{received} \]

The condition states that the sent date is less than or equal to the received date.

Axiomatic descriptions can be made generic in the same manner as schemas. The following example is taken from [Spivey 89] and defines functions giving the first and second elements of an arbitrary ordered pair.

\[ \text{first } (X \times Y) \rightarrow X \]
\[ \text{second } (X \times Y) \rightarrow Y \]

\[ \forall x : X; y : Y \cdot \]
\[ \text{first}(x,y) = x \land \text{second}(x,y) = y \]

The cross product symbol \( \times \) is used to combine sets of ordered pairs. Thus, \( X \times Y \) is the set of all ordered pairs whose first element comes from \( X \) and second element comes from \( Y \). The total function symbol \( \rightarrow \) is used to define a class of total functions. Thus, \( X \rightarrow Y \) is the set of all total functions from \( X \) to \( Y \).
5.3 Graphs, Categories and Diagrams

This section introduces the basic notions of graph, category and diagram. It contains the following sub-sections:

Basic Definitions
Examples

The first section introduces the basic definitions of graph, category and diagram, motivating their introduction with pictures drawn in the process of computer system design. The second section introduces examples of categories based upon sets.

5.3.1 Basic Definitions

This section introduces the basic definitions of graph, category and diagram. The motivation for introducing these notions is based upon pictures that may be drawn in computer system design.

In computer system design, pictures are often drawn with objects and arrows linking the objects. For example, objects might be used to indicate different possible designs, and arrows used to indicate design preferences as in the Figure 5.3.

![Figure 5.3 An Ordering of Design Preferences](image-url)
In Figure 5.3, for example, design e is preferred to both designs c and b, design c is equivalent to design d; and there is no comparison between design b and c.

Pictures are also used to describe the behaviour of systems, where objects represent states and arrows the transition from one state to another, as in the Figure 5.4:

![Figure 5.4 A State Transition Diagram](image)

Figure 5.4 represents the design for a simple display for a computer information system. The objects A, B, D, E represent the different screens and the arrows A₁, B₁, B₂, D₁ represent the possible transitions from screen to screen.

In the field of mathematics, such pictures are known as graphs. The concept of a graph over object and arrow classes has the following formal definition expressed as a Z schema:

\[
\text{Graph} \langle \text{CObjects, CArrows} \rangle
\]

\begin{align*}
\text{objects} & : \mathcal{P} \text{CObjects} \\
\text{arrows} & : \mathcal{P} \text{CArrows} \\
\text{source} & : \text{arrows} \to \text{objects} \\
\text{target} & : \text{arrows} \to \text{objects} \\
\rightarrow & : \text{arrows} \leftrightarrow (\text{objects} \leftrightarrow \text{objects})
\end{align*}

\[
\forall f : \text{arrows}; \ a,b : \text{objects} \bullet \\
f:a \rightarrow b \\
\leftrightarrow \\
(source(f) = a \land target(f) = b)
\]
Graphs often present information implicitly. For example, consider Figure 5.3. It is implicit that since design c is equivalent to design d, then e must be better than d. Representing all such situations, one obtains Figure 5.5.

In addition, consider the state transition diagram of Figure 5.4. Clearly, there is a composite transition from A to D which involves first the transition A1 followed by the transition B1. This transition can be denoted B1°A1. In fact, adding other such transitions gives Figure 5.6.

Clearly, there is also an identity between objects in both diagrams. In the ordering, it is clear that every design must be as good as itself. In the state transition diagram, the identity may be thought of as the 'do nothing' transition.

These discussions motivate the definition of a category as a graph with additional structure, which provides an identity arrow for each object and a
composition operation for each pair of arrows. Formally, a category is defined with a generic schema. The schema is generic over the classes of COBjects and CArrows. It consists of a Graph defined on the generic classes, together with an identity function which maps objects to arrows, and an infix composition operation which maps pairs of arrows to arrows:

\[
\begin{align*}
\text{Category } [\text{COBjects,CArrows}] \cong \\
\text{Graph}[\text{COBjects,CArrows}] \\
id : \text{objects } \rightarrow \text{arrows} \\
\circ : \text{arrows } \times \text{arrows } \rightarrow \text{arrows}
\end{align*}
\]

\[
\forall \ a,b,c,d : \text{objects} \ ; f,g,h : \text{arrows} \cdot \\
(f:a \rightarrow b \land g:b \rightarrow c \land h:c \rightarrow d) \\
\Rightarrow \\
(id(b) \circ f = f \\
\land \\
g \circ id(b) = g \\
\land \\
h \circ (f \circ g) = (h \circ g) \circ f)
\]

The condition states that:

- Right-hand composition with the identity has no effect.
- Left-hand composition with the identity has no effect.
- The composition operation is associative.
The ordering example given above is an instance of a special kind of category called a pre-order category, or just pre-ordering. Formally, this category can be defined using a generic schema. The schema is generic over the class CObjects. The pre-ordering consists of a category defined over CObjects and pairs of CObjects, together with three relations between arrows, less than, equals, and strictly less than.

\[ \text{PreOrdering} \left[ \text{CObjects} \right] \]

\begin{align*}
\text{Category} & : \text{CObjects} \times \text{CObjects} \\
\_\leq & : \text{arrows} \leftrightarrow \text{arrows} \\
\_\_\_\_ = & : \text{arrows} \leftrightarrow \text{arrows} \\
\_\_\_\_ \_\_ & : \text{arrows} \leftrightarrow \text{arrows} \\
\end{align*}

\begin{align*}
\forall \ a, b : \text{objects} \cdot f : \text{arrows} \cdot \\
f = (a, b) \quad \Rightarrow \\
\quad (\text{source}(f) = a) \land \text{target}(f) = b)
\end{align*}

\begin{align*}
\forall \ a, b : \text{objects} \cdot \\
(\exists f : \text{arrows} \cdot f : a \to b) \quad \Rightarrow \\
a \leq b
\end{align*}

\begin{align*}
\forall \ a, b : \text{objects} \cdot \\
(a \leq b \land b \leq a) \quad \Rightarrow \\
a = b
\end{align*}

\begin{align*}
\forall \ a, b, c : \text{objects} \cdot \\
(a \leq b \land \neg(a = b)) \quad \Rightarrow \\
a < b
\end{align*}
The conditions state that:

- The source (respectively target) of an arrow is the first (respectively second) element of the pair.
- If there is an arrow from one object to another, then the first object is less than or equal to the second.
- Two objects are equal, when they are less than or equal to each other.
- One object is strictly less than another, if the less than relationship holds and they are not equal.

It is often difficult to talk about categories using logical formulas. Consider, for example, describing the preceding ordering and state transition diagrams using only text to list all the objects, arrows, source, target, identity and composition mappings. For this reason, commutative diagrams are often used to describe properties of categories. The concept of a diagram can be defined with a schema generic over the classes CObjects and CArrows. It consists of a graph and a category over the objects and arrows together with two mappings, objectLabel, which labels graph objects with category objects, and arrowLabel which labels graph arrows with category arrows.

```
Diagram [CObjects, CArrows]

graph : Graph[CObjects, CArrows]
category : Category[CObjects, CArrows]
objectLabel : graph.objects → category.objects
arrowLabel : graph.arrows → category.arrows
```

∀ a, b : graph.objects ; f : graph.arrows •
  f:a → graph b
  ⇒
  arrowLabel(f) : objectLabel(a) → category objectLabel(b)

The condition states that object and arrow labelling preserves arrows.
More simply, a diagram is a graph whose objects and arrows are labelled with the objects and arrows of a category, so that the structure of the category is preserved. Such diagrams can be used to assert that certain equality relationships hold between arrows. Such diagrams are called commutative diagrams. The class of commutative diagrams can be defined with a schema generic over classes CObjects and CArrows. It consists of a diagram over these objects that obeys an extra condition.

\[ \text{CommutativeDiagram} \subseteq \text{Diagram} \]

\[ \forall \ a,b,c : \text{graph.objects} ; f,g,h: \text{graph.arrows} \cdot (f:a \to b \land g:b \to c \land h:a \to c) \]

\[ \Rightarrow \text{arrowLabel}(g) \circ \text{arrowLabel}(f) = \text{arrowLabel}(h) \]

The condition states that for any triangle of arrows in the diagram, the composition of the two sides is equal to the other side.
The conditions required of the identity and composition operations of a category can be described by insisting that certain diagrams commute. In what follows, diagrams will be used as part of the predicate component of $Z$ schemas. The following is the definition of a category using commuting diagrams:

\[
\text{Category } [\text{CObjects}, \text{CArrows}]
\]

\[
\text{Graph}[\text{CObjects}, \text{CArrows}]
\]

\[
\begin{align*}
\text{id} & : \text{objects} \rightarrow \text{arrows} \\
\circ & : \text{arrows} \times \text{arrows} \rightarrow \text{arrows}
\end{align*}
\]

\[
\forall a,b,c,d : \text{objects} ; f,g,h : \text{arrows} \cdot
\]

In this section, the basic concepts of graph, category and commutative diagram have been derived from examples of simple pictures that might be used in the design of interactive systems. In the next section, more examples of categories are presented.
5.3.2 Examples

This section introduces more examples of categories based upon the notion of a set.

The most simple class of examples of a category are the discrete categories. A discrete category is a category whose only arrows are the identity arrows. Thus, any set \( S \) can be considered as a discrete category \( C \) such that, \( S = C.\text{objects} \). Formally, the function DiscreteCategory is defined over a universe \( U \). The function assigns to the subsets of \( U \) a category whose objects and arrows sets are subsets of \( U \):

\[
\text{DiscreteCategory} : \mathcal{P} U \to \text{Category}[\mathcal{P} U][\mathcal{P} U]
\]

\[
\forall X : \mathcal{P} U \cdot
\begin{align*}
\text{DiscreteCategory}(X).\text{objects} &= X \\
\wedge \\
\text{DiscreteCategory}(X).\text{arrows} &= X \\
\wedge \\
\forall a : \text{DiscreteCategory}(X).\text{objects} \cdot
\end{align*}
\]

\[
\text{id}(a) = a
\]

The condition states that for any subset \( X \) of the universe \( U \), the following hold:

- The objects of the discrete category are the elements of the set \( X \).
- The arrows of the discrete category are the elements of the set \( X \).
- The identity arrow of any object \( a \) of the discrete category is the arrow \( a \).

The power set of a set, that is the set of all subsets of a set, can be used to form categories in a variety of ways. The arrows can, for example, be relations, partial functions or functions.
In order to define these categories, it is convenient to set up the different classes of arrows that might be used. Consider first the relations between the power sets of some set $X$. The relations have three attributes: source, relation and target.

\[
\text{RelationsOver}^\mathcal{P}[X]
\]

\[
\begin{array}{l}
\text{source} : \mathcal{P} X \\
\text{relation} : X \leftrightarrow X \\
\text{target} : \mathcal{P} X \\
\end{array}
\]

\[
\begin{array}{l}
\text{dom}(\text{relation}) \subseteq \text{source} \\
\text{ran}(\text{relation}) \subseteq \text{target} \\
\end{array}
\]

The condition states that the domain of the relation must be contained within the source, and the range of the relation must be contained within the target.

Consider next the partial functions between the power sets of some set $X$. The partial functions have three attributes: source, function and target.

\[
\text{PartialFunctionsOver}^\mathcal{P}[X]
\]

\[
\begin{array}{l}
\text{source} : \mathcal{P} X \\
\text{function} : X \leftrightarrow X \\
\text{target} : \mathcal{P} X \\
\end{array}
\]

\[
\begin{array}{l}
\text{dom}(\text{function}) \subseteq \text{source} \\
\text{ran}(\text{function}) \subseteq \text{target} \\
\end{array}
\]

The condition states that the domain of the relation must be contained within the source, and the range of the relation must be contained within the target.

It can be seen that the partial functions form a special case of the relations. In a similar fashion, the total functions form a special case of the partial
functions. This situation is represented in the following Z schema. The total functions over some power set consists of the partial functions that obey an extra condition:

\[ \text{TotalFunctionsOver}^\mathcal{P}[X] \overline{\text{PartialFunctionsOver}}^\mathcal{P}[X] \]

\[ \text{dom(function)} = \text{source} \]

The condition asserts that the domain of the function is equal to source.
The category of total functions over the power set of some set can now be
defined using a generic axiomatic description. Formally, the function
CategoryOfTotalFunctionsOver\(\mathcal{P}\) is defined over a universe \(U\). The function
assigns to the subsets of \(U\) a category, whose objects set is a subset of the
subsets of \(U\), and whose arrow set is a subset of the total functions over
subsets of \(U\):

\[
\text{CategoryOfTotalFunctionsOver}\mathcal{P}:
\mathcal{P}U \rightarrow \text{Category}[\mathcal{P}(\mathcal{P}U), \mathcal{P}\text{TotalFunctionsOver}\mathcal{P}[U]]
\]

\[
\forall X : \mathcal{P}U \bullet
\text{CategoryOfTotalFunctionsOver}\mathcal{P}(X).\text{objects} = \mathcal{P}X
\]

\[
\text{CategoryOfTotalFunctionsOver}\mathcal{P}(X).\text{arrows}
= \text{TotalFunctionsOver}\mathcal{P}[X]
\]

\[
(\forall f : \text{CategoryOfTotalFunctionsOver}\mathcal{P}(X).\text{arrows} \bullet
\text{source}(f) = f.\text{source}
\wedge
\text{target}(f) = f.\text{target})
\]

\[
(\forall a : \text{CategoryOfTotalFunctionsOver}\mathcal{P}(X).\text{objects} \bullet
\forall x : a \bullet
\text{id}(a).\text{function}(x) = x)
\]

\[
(\forall a,b,c : \text{CategoryOfTotalFunctionsOver}\mathcal{P}(X).\text{objects} \bullet
\forall f,g : \text{CategoryOfTotalFunctionsOver}\mathcal{P}(X).\text{arrows} \bullet
(f:a \rightarrow b \wedge g:b \rightarrow c)
\Rightarrow
\quad g \circ f = g.\text{function} \circ f.\text{function})
\]

121
The condition states that for any subset $X$ of the universe $U$:

- The objects of the category are the elements of the power set of $X$.
- The arrows of the category are the total functions over the power set of $X$.
- The source (respectively target) mapping is defined for each arrow, $f$, as the source (respectively target) attribute of $f$.
- The identity arrow is defined as the identity function.
- The composition of arrows is the composition of functions.
The category of partial functions over the power set of some set can now be defined in an analogous manner. Formally, the function $\text{CategoryOfPartialFunctionsOver}^P$ is defined over a universe $U$. The function assigns to the subsets of $U$ a category whose objects set is a subset of the subsets of $U$, and whose arrow set is a subset of the partial functions over subsets of $U$:

\[ \text{CategoryOfPartialFunctionsOver}^P : \mathcal{P} U \to \text{Category}[\mathcal{P} (\mathcal{P} U), \mathcal{P} \text{PartialFunctionsOver}^P [U]] \]

\[
\forall X : \mathcal{P} U \cdot
\text{CategoryOfPartialFunctionsOver}^P (X).\text{objects} = \mathcal{P} X
\]

\[
\text{CategoryOfPartialFunctionsOver}^P (X).\text{arrows} = \text{PartialFunctionsOver}^P [X]
\]

\[
(\forall f : \text{CategoryOfPartialFunctionsOver}^P (X).\text{arrows} \cdot
\text{source}(f) = f.\text{source}
\land
\text{target}(f) = f.\text{target})
\]

\[
(\forall a : \text{CategoryOfPartialFunctionsOver}^P (X).\text{objects} \cdot
\forall x : a \cdot
\text{id}(a).\text{function}(x) = x)
\]

\[
(\forall a,b,c : \text{CategoryOfPartialFunctionsOver}^P (X).\text{objects} \cdot
\forall f,g : \text{CategoryOfPartialFunctionsOver}^P (X).\text{arrows} \cdot
(f : a \to b \land g : b \to c)
\Rightarrow
\text{g}^\circ f = g.\text{function}^\circ f.\text{function})
\]
The condition states that for any subset X of the universe U:

- The objects of the category is the power set of X.
- The arrows of the category are the partial functions over the power set of X.
- The source (respectively target) mapping is defined for each arrow, f, as the source (respectively target) attribute of f.
- The identity arrow is defined as the identity function.
- The composition of arrows is the composition of functions.
The category of relations functions over the power set of some set can now be defined in an analogous manner. Formally the function \( \text{CategoryOfRelationsOver}_P \) is defined over a universe \( U \). The function assigns to the subsets of \( U \) a category whose objects set is a subset of the subsets of \( U \), and whose arrow set is a subset of the relations over subsets of \( U \):

\[
\text{CategoryOfRelationsOver}_P: \quad \mathcal{P} U \to \text{Category}[\mathcal{P}(\mathcal{P} U), \mathcal{P} \text{RelationsOver}_P[U]]
\]

\[
\forall X: \mathcal{P} U \cdot \\
\text{CategoryOfRelationsOver}_P(X).\text{objects} = \mathcal{P} X
\]

\[
\text{CategoryOfRelationsOver}_P(X).\text{arrows} = \text{RelationsOver}[\mathcal{P}[X]]
\]

\( (\forall f: \text{CategoryOfRelationsOver}_P(X).\text{arrows} \cdot \\
\text{source}(f) = f.\text{source} \land \text{target}(f) = f.\text{target}) \)

\( (\forall a: \text{CategoryOfRelationsOver}_P(X).\text{objects} \cdot \\
\text{id}(a).\text{relation} = \{(x,y): a \mid x=y\}) \)

\( (\forall a,b,c : \text{CategoryOfRelationsOver}_P(X).\text{objects} \cdot \\
\forall f,g: \text{CategoryOfRelationsOver}_P(X).\text{arrows} \cdot \\
(f:a \to b \land g:b \to c) \Rightarrow g \circ f = g.\text{relation} \circ f.\text{relation}) \)
The condition states that for any subset X of the universe U:

- The objects of the category is the power set of X.
- The arrows of the category are the relations over the power set of X.
- The source (respectively target) mapping is defined for each arrow, f, as the source (respectively target) attribute of f.
- The identity arrow is defined as the identity function.
- The composition of arrows is the composition of relations.

Other examples of categories are presented in the sections that follow.

5.4 Monics, Epics and Isos

This section introduces arrows with certain properties. These are monic, epic and iso arrows.

Consider first a state transition diagram or diagrams describing some of the behaviour of a text or graphics editor. Many such editors would have a paste function which adds new text or diagrams to the picture. The paste function is rather special, since it does not disturb the existing structure of the picture or text, but rather adds to it. In other words, all the existing aspects of the text or picture are still there. This situation can be described in terms of commutative diagrams. Consider the following diagram:

If the above diagram commutes, then op1 and op2 must have the same effect. In other words, they must be equal arrows. This equality follows from the fact that a paste operation only adds to the structure of the object edited and does not change any of the existing structure. Thus, it cannot undo any of the differences between the effects of op1 and the effects of op2. In general, the kind of arrow of which paste is an instance, is called a monic. It is defined
formally with the following axiomatic description, which is generic over the classes CObjects and CArrows. The infix construct isMonicIn is defined as a binary relation between arrows and a category defined over that class of arrows:

\[ \text{isMonicIn}_\text{C} : \text{CArrows} \leftrightarrow \text{Category}[\text{CObjects,CArrows}] \]

\[ \forall f : \text{CArrows}; C : \text{Category}[\text{CObjects,CArrows}] \cdot \\
    f \text{ isMonicIn } C \\
    \Leftrightarrow \\
    (f \in C.\text{arrows} \\
     \land \\
     (\forall a,b,c : C.\text{objects}; g,h : C.\text{arrows} \cdot \\
     \begin{array}{c}
     c \\
     \downarrow \quad g \quad \downarrow \\
     h \\
     b \\
     \downarrow \quad f \\
     \end{array} \\
     \text{commutes} \\
     \Rightarrow \\
     g = h) \]

The condition states that an arrow \( f \) is monic in a category \( C \), exactly when \( f \) is an arrow of \( C \) and \( f \) is left-cancellable.
Note that a cut operation is clearly not monic, since a cut could undo the differences between previous operations. The cut, however, does have a different property. The cut adds nothing new to the structure being edited. In other words, after a cut operation, there are no aspects of the text or picture, that were not there before. Again, the situation can be described in terms of commutative diagrams. Consider the following diagram:

\[
\begin{array}{c}
S_1 \xrightarrow{\text{Cut}} S_2 \\
\downarrow \text{Cut} \\
S_2 \xrightarrow{\text{op}_2} S_3 \\
\downarrow \text{op}_1 \\
S_3
\end{array}
\]

If the above diagram commutes then \( \text{op}_1 \) and \( \text{op}_2 \) must have the same effect. In other words, they must be equal arrows. This equality follows from the fact that a cut operation only removes some of the structure of the object edited and does not add to the existing structure. Thus, the operations \( \text{op}_1 \) and \( \text{op}_2 \) could be applied to the object edited without the cut, and must have the same effect. In general, the kind of arrow of which cut is an instance is called an epic. It is defined formally with the following axiomatic description which is generic over the classes COObjects and CArrows. The infix construct isEpicIn is defined as a binary relation between arrows and a category defined over that class of arrows:
The condition states that an arrow $f$ is epic in a category $C$, exactly when $f$ is an arrow of $C$ and $f$ is right-cancellable.

An example of an operation that is neither monic nor epic is the search and replace function.

It is important to note that for many text and graphics editor implementations operations, such as cut and paste, might be quite sophisticated, and it cannot be guaranteed that they will always be epic or monic. For example, on many text editors, a paste may be carried out when there is text selected. The effect of this paste is to replace the selected text with the pasted text. In such circumstances, the paste operation cannot be considered as monic.

Another notion which can be expressed in state transition diagrams is that of reversibility. If an operation $op_1$ is reversible, then there must be some operation $op'$ such that $op'$ can reverse $op$. In other words, performing $op$
followed by op' is the same as doing nothing. Put in diagrammatic form an operation op is reversible if there is an operation op' such that the following diagram commutes:

\[ \begin{array}{ccc}
S1 & \xrightarrow{\text{op}} & S2 \\
\downarrow{\text{id}(S1)} & & \downarrow{\text{op}'} \\
S1 & & 
\end{array} \]

In category theory, the generalisation of a reversible arrow is called an iso. It is defined formally with the following axiomatic description which is generic over the classes CObjects and CArrows. The infix construct isIsoIn is defined as a binary relation between arrows and a category defined over that class of arrows:
The condition states that an arrow $f$ is iso in a category $C$, exactly when $f$ is an arrow of $C$, and $f$ has an inverse such that $f$ composed with its inverse is the identity function.

In any pre-order, iso arrows link equivalent objects. Thus, in the design pre-order example of the previous section, the iso arrows link equivalent designs. Iso arrows are always monic and epic, but the converse does not hold. In any pre-order, such as the orderings of designs in the previous section (Figure 5.3), since there is at most one arrow between objects, every arrow is both monic and epic, but not all arrows are iso. In a discrete category, the only arrows are the identity arrows. These arrows are iso, and hence both epic and monic. For the category of relations, partial functions and total functions over some set, the monics are precisely the injective total functions, the epics are the surjective total functions, and the isos are the bijections.
5.5 Functors and Categories of Categories

This section considers how categories of categories can be constructed. It contains the following sections:

- Functors
- Sub Categories
- Categories of Categories

The first section introduces the notion of a functor motivating its introduction by considering state transition diagrams. The next section considers sub­categories. The final section considers categories of categories.

5.5.1 Functors

Consider two state transition diagrams:

These diagrams both might represent the behaviour of some computer menu system. The second diagram can be seen as an abstraction of the first. It is possible to demonstrate the abstraction by defining an operation which maps the first diagram onto the second, while preserving the structure of the first diagram. This operation is illustrated in Figure 5.7:
Generalising the notion of such structure preserving mappings gives rise to the definition of a functor. The notion of a functor is defined formally with the following Z schema. It is generic over classes of CObjects and CArrows. It has four components. The first two represent the source and target category of the functor. The last two represent the mappings from the objects and arrows of the source category to the objects and arrows of the target category.

The conditions state that:

- Arrows are preserved
- Identities are preserved
- Composition of arrows is preserved
Functor $\mathbb{F} = \mathbb{F}[\text{CObjects}, \text{CArrows}]$

- **sourceCategory** : Category[$\text{CObjects, CArrows}$]
- **targetCategory** : Category[$\text{CObjects, CArrows}$]
- **objectMap** : sourceCategory.objects $\rightarrow$ targetCategory.objects
- **arrowMap** : sourceCategory.arrows $\rightarrow$ targetCategory.arrows

\[
\forall a, b : \text{sourceCategory.objects}; f : \text{sourceCategory.arrows} \\
\quad f : a \rightarrow b \\
\Rightarrow \\
\quad \text{arrowMap}(f) : \text{objectMap}(a) \rightarrow \text{objectMap}(b)
\]

\[
\forall a : \text{sourceCategory.objects} \\
\quad \text{arrowMap}(\text{sourceCategory.id}(a)) \\
\quad = \\
\quad \text{targetCategory.id}(\text{objectMap}(a))
\]

\[
\forall a, b, c : \text{sourceCategory.objects}; f, g, h : \text{sourceCategory.arrows} \\
\quad \text{commutes in sourceCategory} \\
\Rightarrow \\
\quad \text{commutes in targetCategory}
\]

In many ways, functors are similar to arrows. There are identity functors and, for functors with the same source and target, it is possible to define functor
composition. These notions are defined formally in the following generic
axiomatic definition. The definition is generic over the classes CObjects and
CArrows. The identity functor is defined as a function from categories to
functors. Functor composition is defined as a partial function from the cross-
product of functors to functors:

\[ \text{idFunctor : Category[CObjects,CArrows] \rightarrow Functor[CObjects,CArrows]} \]

\[ \text{o} \quad \theta \quad (\text{Functor[CObjects,CArrows]} \times \text{Functor[CObjects,CArrows]}) \]

\[ \rightarrow \text{Functor[CObjects,CArrows]} \]

\[ \forall \ C : \text{Category[CObjectsCArrows]} \]

\[ \text{idFunctor(C).sourceCategory} = \text{C} \]

\[ \wedge \]

\[ \text{idFunctor(C).targetCategory} = \text{C} \]

\[ \wedge \]

\[ (\forall \ a : \text{C.objects} \]

\[ \text{idFunctor(C).objectMap(a)} = a \]

\[ \wedge \]

\[ (\forall \ f : \text{C.arrows} \]

\[ \text{idFunctor(C).arrowMap(f)} = f \]

\[ \text{dom(\text{o})} = \{F,F' : \text{Functors[CObjects,CArrows]} \mid \]

\[ \text{F.sourceCategory} = \text{F.targetCategory} \}

\[ \forall \ F,F' : \text{Functor[CObjects,CArrows]} \]

\[ \text{F}^{\circ}F'.\text{sourceCategory} = F'.\text{sourceCategory} \]

\[ \wedge \]

\[ \text{F}^{\circ}F'.\text{targetCategory} = F'.\text{targetCategory} \]

\[ \wedge \]

\[ \text{F}^{\circ}F'.\text{objectMap} = \text{F.objectMap}^{\circ}F'.\text{objectMap} \]

\[ \wedge \]

\[ \text{F}^{\circ}F'.\text{arrowMap} = \text{F.arrowMap}^{\circ}F'.\text{arrowMap} \]

The first condition states a functor is an identity functor of some category
exactly when each of the following holds:
• The source category of the functor is the category
• The target category of the functor is the category
• The object map is an identity function
• The arrow map is an identity function

The second condition states that the domain of the composition operation is the set of pairs of functors, in which the source category of the first functor is the same as the target category of the second.

The final condition states that for any composition:

• The source category is the source category of the second functor
• The target category is the target category of the first functor
• The object map is the composition of object maps
• The arrow map is the composition of arrow maps

5.5.2 Sub-Categories

Consider the two pre-orderings given by the following diagrams:

Pre-Ordering 1

Pre-Ordering 2

Pre-Ordering 1 has all the objects and arrows that are in Pre-Ordering 2. Pre-Ordering 1 can said to be a sub-ordering of Pre-Ordering 2. Generalising to categories, the notion of a sub-category is obtained.

The notion of sub-category is defined formally with the following generic axiomatic definition. The definition is generic over the classes of CObjects and CArrows. It defines an infix relation between categories:
The condition states that a category is a subset of another category precisely when the sets of objects and arrows are subsets.

For any set $S$:

\[
\text{CategoryOfTotalFunctionsOver}_P(S) \subseteq \text{CategoryOfPartialFunctionsOver}_P(S)
\]

\[
\text{CategoryOfPartialFunctionsOver}_P(S) \subseteq \text{CategoryOfRelationsOver}_P(S)
\]

The notion of a sub-category will be used in Chapter 6, in order to define the relationship between cost, performance and benefit orderings.
5.5.3 Categories of Categories

Given a set \( S \) the power set is the set of all subsets of \( S \). In a similar manner, it is possible to define a number of different kinds of category of categories. The one considered here is a category whose objects are categories, and whose arrows are functors between categories. Such a category is called the category of functors.

Formally this category of categories can be defined with a generic axiomatic construct. The generic classes are \( \text{CObjects} \) and \( \text{CArrows} \). The category of functors is a function from the cross-product of the subsets of \( \text{CObjects} \) and \( \text{CArrows} \) to the categories whose objects are the categories over \( \text{CObjects} \) and \( \text{CArrows} \) and whose arrows are the functors over \( \text{CObjects} \) and \( \text{CArrows} \).

The \( \text{CategoryOfFunctorsOver(M,C)} \) notation is an extension of the Z notation which enables the infix operator (in this case \( ° \)) to be referenced. In general, for any infix operator \( \text{I} \) and constant \( C \),

\[
\_I_C_ \text{ is an abbreviation for } C.\text{I}_
\]

The condition states that for any class of objects \( O \) and arrows \( A \), the category of functors over \( O \) and \( A \) must satisfy the following conditions:

- Its objects are categories whose objects are contained in \( O \) and arrows contained in \( A \).
- Its arrows are the functors between its objects.
- The source of a functor is the source category of the functor.
- The target of a functor is the target category of the functor.
- Identity arrows are identity functors.
- Composition of arrows is composition of functors.
\[ \forall O : \mathcal{P} \text{CObjects}; A : \mathcal{P} \text{CArrows} \bullet \\
\text{CategoryOfFunctorsOver}(O,A).\text{objects} = \{ C : \text{Category}[\text{CObjects},\text{CArrows}] \mid C.\text{object} \subseteq O \land C.\text{arrows} \subseteq A \} \\
\land \text{CategoryOfFunctorsOver}(O,A).\text{arrows} = \\
\{ F : \text{Functor}[\text{CObjects},\text{CArrows}] \mid F.\text{sourceCategory} \in \\
\text{CategoryOfFunctorsOver}(O,A).\text{objects} \\
\land F.\text{targetCategory} \in \\
\text{CategoryOfFunctorsOver}(O,A).\text{objects} \} \\
\land \forall F : \text{CategoryOfFunctorsOver}(O,A).\text{arrows} \bullet \\
\text{CategoryOfFunctorsOver}(O,A).\text{source}(F) = F.\text{sourceCategory} \\
\land \forall F : \text{CategoryOfFunctorsOver}(O,A).\text{arrows} \bullet \\
\text{CategoryOfFunctorsOver}(O,A).\text{target}(F) = F.\text{targetCategory} \\
\land \forall C' : \text{CategoryOfFunctorsOver}(O,A).\text{objects} \bullet \\
\text{CategoryOfFunctorsOver}(O,A).\text{id}(C') = \text{idFunctor}(C') \\
\land \forall F, F' : \text{CategoryOfFunctorsOver}(O,A).\text{arrows} \bullet \\
F.\text{sourceCategory} = F.\text{targetCategory} \\
\Rightarrow \\
F \circ \text{CategoryOfFunctorsOver}(O,A) F' = F \circ F' \]
The category of functors over a category is in many ways similar to the category of total functions over a set. In the same fashion, it is possible to define notions of 'partial functor' and 'relator' and corresponding categories of categories.

A special case of the category of categories over some objects and arrows is the category of pre-orderings over some class of objects. Formally, the category of pre-orderings can be defined with a generic axiomatic construct. The generic class is CObjects. The category of pre-orderings is a function from the sets of CObjects to the categories over the pre-orderings of CObjects and functors over CObjects and pairs of CObjects.

The schema condition states that for any class of objects O, the category of pre-orderings over O must satisfy the following conditions:

- The category of pre-orderings over O is a sub category of the category of functors over O and the pairs of O.
- Its objects are categories whose objects are contained in O.
- Its arrows are the functors between its objects.
5.6 Limits and Co-limits

This section introduces the concepts of limits and co-limits. It contains the following sections:

- Basic Definitions
- Particular Limits and Co-Limits
- Z notation, Limits and Co-limits
The first sub-section introduces the notions of Limit and Co-Limit, motivated by the discussion of partial orderings. The second sub-section considers particular limits and provides examples from categories of sets. The final sub-section argues that the key structuring mechanisms of \( Z \) are means of expressing limits and co-limits.

### 5.6.1 Basic Definitions

Consider the pre-order category of Figure 5.8.

![Figure 5.8 An Example of a Pre-order Category](image)

Consider as well the diagram over this category consisting of just the objects \( a \) and \( b \):

![a -> b](image)

Suppose the pre-order of Figure 5.8 represents an ordering of possible designs, then one might wish to consider all the designs which are less desired than both \( a \) and \( b \) (Figure 5.9):

![a -> b, c -> d -> e](image)

Generalising this notion to any diagram in any category gives the concept of a cone. A cone is defined as a schema generic over the classes of COObjects and CArrows. It consists of a Diagram together with a cone object, which is an
object of the category of the diagram, and a function called projection which maps graph objects to category arrows:

\[ \text{Cone}_\text{Diagram} [\text{CObjects}, \text{CArrows}] \]

\[
\begin{align*}
\text{Diagram}[\text{CObjects}, \text{CArrows}] \\
\text{coneObject} & : \text{category.objects} \\
\text{projection} & : \text{graph.objects} \rightarrow \text{category.arrows}
\end{align*}
\]

\[
\forall d : \text{graph.objects} \quad \text{projection}(d) : \text{objectLabel}(d) \rightarrow \text{coneObject}
\]

\[
\forall d,d' : \text{graph.objects} \quad \forall g : \text{graph.arrows} \\
\quad g : d \rightarrow d' \\
\Rightarrow \\
\quad \text{projection}(d) \rightarrow \text{projection}(d')
\]

commutes in category
The conditions state that:

- For any object of the diagram the projection of that object is an arrow to that object from the coneObject.
- The arrows of the diagram commute with projections.

Having defined the concept of a cone, it is possible to define the function, cones, which gives all the cones of a diagram. Formally, cones can be defined with an axiomatic construct, generic over the classes CObjects and CArrows. Cones is a function from the diagrams over the generic classes to the set of cones over the generic classes:

\[ \text{cones} : \text{Diagram[CObjects,CArrows]} \rightarrow \mathcal{P} \text{Cone[CObjects,CArrows]} \]

\[
\forall \, D : \text{Diagram[CObjects,CArrows]}
\quad \text{cones}(D) =
\quad \{ C : \text{Cone[CObjects,CArrows]} \mid
\quad \quad \text{D.graph} = \text{limit}(D).\text{graph}
\quad \land \quad \text{D.category} = \text{limit}(D).\text{category}
\quad \land \quad \text{D.objectLabel} = \text{limit}(D).\text{objectLabel}
\quad \land \quad \text{D.arrowLabel} = \text{limit}(D).\text{arrowLabel}) \}
\]

The condition states that the cones of a diagram is precisely the set of all cones with that diagram.

Suppose that in a design ordering, one wants to find the most desired option that is less than both a and b. Clearly, in Figure 5.8 this is c, since e and d are both less desired than c. This can be seen from Figure 5.10:
Since the diagrams in Figure 5.10 are diagrams over a pre-order, which by definition, has at most one arrow between any two objects, they must commute.

Generalising this notion to any class of diagrams, one obtains the concept of a limit. Formally, the notion of a limit can be defined with an axiomatic construct, generic over the classes CObjects and CArrows. The limit is a partial function from the diagrams over the generic classes to the cones over the generic classes:

\[
\text{limit} : \text{Diagram}[\text{CObjects},\text{CArrows}] \leftrightarrow \text{Cone}[\text{CObjects},\text{CArrows}]
\]
The condition states that for all diagrams:

- The limit of the diagram is a cone of the diagram.
- For any other cone, there is a unique arrow from that cone object to the cone object of the limit, so that the projections commute.

It is important to note that limits, and indeed cones, need not exist. For example, in the following pre-order:

```
  a  b
  \  /\
  \|/ \\d--e
```

the diagram consisting of just the objects a and b has two cones, but no limit, and the diagram consisting of just objects d and e has no cones at all.

By reversing the arrows in the definition of a cone, one obtains the concept of a co-cone. A co-cone is defined as a schema generic over the classes of CObjects and CArrows. It consists of a diagram together with a co-cone object, which is an object of the category of the diagram, and a function called injection which maps graph objects to category arrows.

The conditions state that:

- For any object of the diagram the injection of that object is an arrow from that object to the co-cone object.
- The arrows of the diagram commute with injections.
In the same manner as for cone, it is possible to define the function co-cones which gives all the co-cones of a diagram. Formally, co-cones can be defined with an axiomatic construct generic over the classes COObjects and CARrows. Co-cones is a function from the diagrams over the generic classes to the set of co-cones over the generic classes:
The condition states that the co-cones of a diagram is precisely the set of all co-cones with that diagram.

In a pre-order category representing designs, a co-cone of a diagram of two objects a and b is any design which is better than both a and b.

The reversing of arrows can also be used to give the definition of a co-limit.
The condition states that for all diagrams:

- The co-limit of the diagram is a co-cone of the diagram.
- For any other co-cone, there is a unique arrow to that co-cone object from the co-cone object of the limit, so that the injections commute.

In a pre-order category of designs, the co-limit of a diagram consisting of just the objects a and b is the least desired design which is more desired than both a and b.
5.6.2 Particular Limits and Co-limits

Different forms of diagrams give rise to specifically named limits and co-limits. The following table defines names for the limits and co-limits of certain diagrams:

<table>
<thead>
<tr>
<th>Diagram</th>
<th>Limit</th>
<th>Co-Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty Diagram</td>
<td>Terminal Object</td>
<td>Initial Object</td>
</tr>
<tr>
<td>a</td>
<td>Product</td>
<td>Co-Product</td>
</tr>
<tr>
<td>b</td>
<td>Equaliser</td>
<td>Co-Equaliser</td>
</tr>
<tr>
<td>a [\rightarrow] b</td>
<td>Pullback</td>
<td></td>
</tr>
<tr>
<td>a [\rightarrow] c [\leftarrow] g [\rightarrow] b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b [\leftarrow] a [\rightarrow] g [\rightarrow] c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let $S$ be a set and let $C$ be the category of total functions over $\mathcal{P}S$. Then the empty set is an initial object and the singleton sets are terminal objects. In the same category, consider the diagram:

$S_1 \xrightarrow{f} S_2 \xleftarrow{g} S_3$

where $f(a) = a$ and $g(b) = b$, then the pullback of this diagram is the *intersection* of sets $S_1$ and $S_2$, that is the set of all elements that belong to $S_1$ and $S_2$, denoted, $S_1 \cap S_2$. In the same category, consider the diagram:

$S_1 \xleftarrow{f} S_2 \xrightarrow{g} S_3$

where $f(a) = a$ and $g(b) = b$, then the pushout of this diagram is the *union* of sets $S_1$ and $S_2$, that is the set of all elements that belong to $S_1$ or $S_2$ denoted, $S_1 \cup S_2$. 
Let $S$ be a set and consider the category of functions over $\mathcal{P}(S \cup (S \times S))$. Let $S_1, S_2 : \mathcal{P}S$. Then the product of the diagram:

\[
\begin{array}{cc}
S_1 & \rightarrow & S_2 \\
\end{array}
\]

Is precisely the cross product $S_1 \times S_2$.

Since it is possible to construct categories of categories, the notions of limit and co-limit can be applied to categories. In particular, we can define products for pre-order categories. The definition takes the form of an axiomatic schema defined over two classes $X$ and $Y$. The cross-product is a function from the product of the pre-orderings to the pre-ordering of the products:

\[
\begin{array}{l}
\forall \, px : \text{PreOrdering}[X]; \, py : \text{PreOrdering}[Y] \\
\forall \, x, x' : px; \, y, y' : py \\
\quad (x, y) \leq (px \times py) (x', y') \\
\quad (x \leq px \land y \leq py) \\
\end{array}
\]

The condition states that the ordering relationship of a product of two pre-orders holds if, and only if, the ordering relationship holds for each component.

The notion of a cross-product of pre-orderings will be used in Chapter 6 to define the relationship between cost, performance and benefit orderings.
5.6.3 Z Notation, Limits and Co-limits

A key structuring mechanism in the Z notation is the schema. Essentially, a schema defines a limit. For any schema of the following form:

```
SchemaName[GenericType1,...,GenericTypeK]
```

```
identifier1 :Type1

......

identifierN :TypeN
```

```
predicate1

......

predicateM
```

The schema specifies the limit, in some category, of a diagram whose objects are the types Type1...TypeN, and whose arrows are represented by the predicates predicate1....predicateM. The identifiers identifier1....identifierN represent the projections.

The other major structuring mechanism in Z is the axiomatic definition, which is a mechanism for constructing co-products.

As Spivey points out [Spivey 88], given that one can construct a category whose limits and co-limits correspond to Z schemas and axiomatic definitions, there is no guarantee that the limit or co-limit will necessarily exist.
5.7 Conclusion

In conclusion, this section summarises the chapter and its achievements and assesses these achievements by the criteria established in the introduction (Chapter 1).

5.7.1 Summary

This chapter provides the basis for the formal framework that is outlined in Chapters 6 to 11. This sub-section summarises the basis of the framework that has been introduced and indicates how it is applied in Chapters 1 to 6.

Section 5.2 introduced the Z notation. The concepts of schema, schema extensions were introduced. The axiomatic description and generic axiomatic description were also introduced. The Z notation is used extensively in Chapters 6 to 10.

Section 5.3 introduced the notions of graph, category and pre-ordering as a special case of categories. Also introduced were the notions of diagrams and commutative diagrams. The examples of categories presented included, discrete categories, categories of relations and total and partial functions. The concept of a pre-order category is employed to capture the notions of benefit, cost and performance orderings in Chapter 6, and categories are employed in Chapters 8 and extensively in Chapter 10.

Section 5.4 introduced the concepts of monics, epics and isos. Section 5.5 introduced the concepts of functors, and categories of categories. In particular, the category of pre-orders was introduced. It is in such a category that the cost and benefit orderings of Chapter 6 are combined to give the performance ordering.

Section 5.6 introduced the concepts of cones and limits and the dual concepts of co-cones and co-limits. Co-cones and co-limits are employed in Chapter 8 in the discussion of how to generalise the notion of agent combination. Special cases of limits and co-limits are introduced. One special case, the product in categories of pre-orderings, is the means by which cost and benefit orderings can be combined in Chapter 6.
5.7.2 Assessment

Recall that Chapter 1 outlined the following criteria for assessing the thesis that are applicable to the framework aspects of the thesis:

- Internal consistency between conception and framework
- External synthesis at the level of the framework
- Knowledge incremented at the level of the framework

The work presented in this section provides the basis upon which the internal consistency between the conception, already outlined, and the framework, presented in Chapters 6 to 11, may be judged.

The main external synthesis of this chapter is the embedding of category theory within the Z notation. As far as is known, the definition of the key concepts of category theory using Z is novel and thus marks an increment to existing knowledge.

Category theory was developed by abstracting across different fields of mathematics. Many of the examples that are traditionally used to motivate the theory involve sophisticated mathematical structures. Examples often include, the category whose objects are groups and arrows are homomorphisms, or the category whose objects are topological spaces and arrows homeomorphisms. The presentation given here has gone to great lengths to avoid such examples, using notions such as design orderings and state transition diagrams. Thus, the presentation of the category theory given here can be seen as an increment of existing knowledge, with the purpose of promoting the use of category theory in work on the design in general and the design of formal languages in particular.
6 Specifying Desired Performance

This chapter develops the aspects of the formal framework concerned with specifying the desired performance of language systems. It is an instantiation of the Conception of Formal Language Engineering developed in Chapter 4. Each of the concepts outlined in Chapter 4 is defined using the Z notation and the categorial extensions developed in Chapter 5.

The first section introduces the benefit requirements component of performance requirements. The second section introduces the cost requirements component of performance requirements. The final section combines the benefit requirements and cost requirements components to create the definition of performance requirements.

A simple paint shop example is employed throughout the chapter. The paint shop example does not purport to represent a formal language design problem that might be of potential interest to some client. Rather, it acts to provide a simple illustration of the concepts being defined.

6.1 Benefit Requirements

This section formally defines the Benefit Requirements of the worksystem using the Z notation. It contains the following sections:

- Work Structures
- Work Domain
- Desired Benefit and the Benefit Ordering
6.1.1 Work Structures

Recall from Chapter 4:

'Work is a transformation of work structures.'

In general, work structures vary extensively and there is little to say formally about their nature. However, any statement of the required benefits of a system will be based upon some class of work structures. Consequently, the definition of the Benefit Requirements of the worksystem outlined in this section, is presented in terms of a generic class of work structures.

The paint shop is concerned with children's bricks. The size and shape of the bricks are considered unimportant. In fact, the only attribute of interest is the colour of the bricks. Consequently, each work structure of the paint shop can be considered as having a single attribute, that of colour. The set of all paint shop work structures can be described using set enumeration.

\[
\text{PSWorkStructures} \equiv \{\text{blue, red, yellow, purple, green, orange, black}\}
\]

6.1.2 Work Domain

Recall from Chapter 4 that:

'The work structure transformations that are of interest to the client who requires a formal language worksystem is called the work domain.'
In line with this conceptual definition, the work domain over a class of work structures is defined formally using the schema notation as follows:

\[
\text{WorkDomain} [\text{WorkStructures}]
\]

\[
\text{domain} : \mathcal{P} \text{WorkStructure} \rightarrow \text{WorkStructure}
\]

A WorkDomain over a set of WorkStructures consists of a single component, the domain, which is a set of partial functions from the WorkStructures to the WorkStructures.

The paint shop domain PSDomain is the set of all the partial functions from PSWorkStructures to PSWorkStructures, formally:

\[
\text{PSDomain} == \\
\text{PSWorkStructure} \rightarrow \text{PSWorkStructure}
\]

Thus, PSWorkdomain is thus defined formally as follows:

\[
\text{PSWorkDomain} == (\text{PSDomain})
\]

PSWorkDomain is an instantiation of the class of work domains over PSWorkStructures. Formally:

\[
\text{PSWorkDomain} \in \text{WorkDomain}[\text{PSWorkStructures}]
\]

A group of example paint shop transformations is now presented. These transformations are used throughout.

\[
\text{TB} == \{(\text{red,blue})\}
\]

Here, red bricks are changed to blue bricks.

\[
\text{TR} == \{(\text{blue,red}),(\text{yellow,red})\}
\]

Here, blue and yellow bricks are changed to red bricks.
TRG == {(blue,red),(yellow,green)}

Here, blue bricks are changed to red bricks and yellow bricks are changed to green bricks.

TGR == {(blue,green),(yellow,red)}

Here, blue bricks are changed to green bricks and yellow bricks are changed to red bricks.

TG == {(blue,green),(yellow,green)}

Here, blue and yellow bricks are changed to green bricks.

Since each transformation of the work domain is a partial function, their range and domain can be determined.

\[
\text{dom}(TB) = \{\text{red}\} \\
\text{dom}(TR) = \{\text{blue}, \text{yellow}\} \\
\text{ran}(TR) = \{\text{red}\} \\
\text{ran}(TRG) = \{\text{red}, \text{green}\}
\]

Each transformation of the work domain can be applied to an object that belongs to its domain to obtain the result of the transformation. Applying TB transformation to a red object will produce a blue object and applying TRG to a yellow object will produce a green object. Formally:

\[
TB(\text{red}) = \text{blue} \\
TRG(\text{yellow}) = \text{green}
\]
6.1.3 Desired Benefits and the Benefit Ordering

Recall from Chapter 4 that

'The benefit ordering represents the ordering of the work domain according to client preferred benefits. The desired benefits of a formal language worksystem is the collection of elements of the work domain whose benefits are desired by the client.'

The benefit requirements of a formal language worksystem consists of the work domain and two further components: the desired benefits and the benefit ordering.

A simple approach to the issues of benefits, costs and performance orderings might be to assign numerical values to the entities which maintain these relationships. For example, in the case of benefits, integers could be assigned to each of the possible transformations of work structures and the benefit ordering constructed on the basis of this numeric value. In the general case, such numerical assignments would amount to an over-specification of the concept of a benefit ordering, since it is possible to envisage that there is some work domain for which particular work structure transformations are incomparable. It seems, therefore, that these relationships would be better based upon some minimal notion of ordering. The concept of a pre-ordering, developed in Chapter 5, provides just such a notion.

The concept of benefit requirements is defined below. The definition has the generic parameters WorkStructures.
The desired benefits component is a sub-set of the work domain representing all the possible transformations of work structures that are desired. The benefit component is a pre-ordering of the work domain.

\[
\text{BenefitRequirements} \subseteq [\text{WorkStructures}]
\]

\[
\text{WorkDomain} [\text{WorkStructures}]
\]

\[
\text{desiredBenefits} : \mathcal{P} \text{ domain}
\]

\[
\text{benefit} : \text{PreOrdering} \text{[domain]}
\]

\[
\forall t,t' : \text{domain} \quad t \subseteq \text{desiredBenefits} \quad \wedge \quad t \preceq_{\text{benefit}} t' \quad \Rightarrow \quad t' \in \text{desiredBenefits}
\]

The condition in the second box states that any work structure transformation in the domain, which is equivalent to or improves upon a desired benefit, is also a desired benefit.

In order to develop a specification of the benefit requirements for the paint shop example, it is necessary to define a set corresponding to each of the two new components. We define the desired benefits component first.

In the example, bricks arrive at the paint shop in one of two initial colours. Formally:

\[
\text{InitialColours} \quad == \quad \{\text{blue,yellow}\}
\]

The paint shop aims to transform the bricks into one of two final colours. Formally:

\[
\text{FinalColours} \quad == \quad \{\text{red,green}\}
\]
The desired benefits for the paint shop can therefore be defined as the set of all transformations of paint shop objects whose domain is the initial colours and whose range is contained in the final colours. Formally:

\[
\text{PSDesiredBenefits} = \{ t : \text{PSDomain} \mid \text{dom}(t) = \text{InitialColours} \land \text{FinalColours} \supseteq \text{ran}(t) \}\
\]

This example is a simple one, in which there are a finite number of transformations, each of which has been presented as an example in the previous sub-section. Consequently, unlike most real situations, we can re-express the desired benefits using set enumeration:

\[
\text{PSDesiredBenefits} = \{ \text{TR,TRG,TGR,TG} \}\
\]

In the paint shop example, the benefit of the transformations is dependant upon the supply of blue bricks, yellow bricks and primary colour paints, together with the demand for red bricks and green bricks. In the example, the supply costs are assumed to be negligible and the demand for red bricks is assumed to be greater than that for green bricks.

Paint shop benefit is specified using an axiomatic descriptions construct:
The conditions state that:

- All transformations which are not desired benefits are of equivalent quality.
- Transformations which have desired benefits and which produce the same set of objects are of equivalent benefit.
- Transformations which have desired benefits are of greater benefit than transformations which do not have desired benefits.
- A transformation which produces only red bricks is of greater benefit than a transformation which produces red and green bricks.
- A transformation which produces red and green bricks is of greater benefit than a transformation which produces only green bricks.

PSBenefit is represented in the Figure 6.1.
Paint shop benefit requirements can now be specified:

```
PSBenefitRequirements ==
(    PSDomain,
    PSDesiredBenefits,
    PSBenefit)
```
6.2 Cost Requirements

This section formally defines the Cost Requirements of the worksystem using the Z notation. It contains the following sections:

- Cost Structures
- Desired Costs and the Cost Ordering

6.2.1 Cost Structures

Recall from Chapter 4, that:

'In transforming work structures, worksystems incur worksystem costs.'

In general, the costs of the worksystem vary according to the kind of language agents that make up the worksystem. For any particular set of cost requirements, only a particular class of all the worksystem costs will be considered. This class of possible costs is defined in the following Z schema, which forms an analogue to the domain.

\[
\text{CostStructures} \rightarrow [\text{WorksystemCosts}]
\]

\[
\begin{align*}
\text{possibleCosts} & : \mathcal{P} \text{WorksystemCosts} \\
\end{align*}
\]

In the paint shop example, it is assumed that the only cost of any concern is the cost of running the worksystem, and this cost can be expressed in terms of units. Thus, in the paint shop example, the worksystem costs can be represented with the set of natural numbers:

\[
\text{PSWorksystemCosts} == \mathbb{N}
\]

The set of costs that are of interest in the paint shop example is all of the natural numbers, formally:

\[
\text{PSPossibleCosts} == \mathbb{N}
\]
The paint shop cost structures are a combination of the paint shop costs and cost operations, formally:

\[ \text{PSCostStructures} == (\text{PSPossibleCosts}) \]

### 6.2.2 Desired Costs and the Cost Ordering

Recall from Chapter 4 that

'The cost ordering represents the ordering of possible work costs according to the client preferred costs. The desired cost of a formal language worksystem is the collection of desired worksystem costs of the system.'

The cost requirements of a formal language worksystem consists of the cost structures together with two additional components, the desired costs and the cost ordering. These concepts are defined below.

The desired costs component is a sub-set of the possibleCosts. The cost component is a partial ordering of the possibleCosts. The condition in the second box states that any cost structure which is equivalent to or improves on a desired cost is also a desired cost.

\[
\begin{array}{l}
\text{CostRequirements}_\text{[WorksystemCosts]} \\
\text{CostStructures}_\text{[WorksystemCosts]} \\
\text{desiredCosts} : \mathcal{P} \text{possibleCosts} \\
\text{cost} : \text{PreOrdering}[\text{desiredCosts}] \\
\forall c,c' : \text{possibleCosts} \quad \bullet \\
\quad c \sqsubseteq \text{desiredCosts} \\
\quad \wedge \\
\quad c \sqsubseteq \text{cost} c' \\
\quad \Rightarrow \\
\quad c \in \text{desiredCosts}
\end{array}
\]
In order to develop a specification of the cost requirements for the paint shop example, it is necessary to define a set corresponding to both components. We define the desired costs component first.

It is assumed that in the paint shop worksystem the desired costs are any cost which is less that 5 units. Therefore, desired costs for the paint shop can be defined as the set of all natural numbers less than 5.

\[
\text{PSDesiredCosts} == \{ n : \mathbb{N} \mid n < 5 \}
\]

In the paint shop example, a worksystem exhibits more desired costs the less units per hour incurred in running the worksystem. Paint shop cost is specified using an axiomatic descriptions construct.

<table>
<thead>
<tr>
<th>PSCost</th>
<th>: PreOrdering[PSPossibleCosts]</th>
</tr>
</thead>
<tbody>
<tr>
<td>∀ c, c' : CostStructures • c ≤_{PSCost} c' \iff c ≥ c'</td>
<td></td>
</tr>
</tbody>
</table>

The conditions state that:

- A cost ordering is the inverse of the natural number ordering.

Paint shop cost requirements can now be specified:

\[
\text{PSCostRequirements} ==
( \text{PSPossibleCosts}, \text{PSDesiredCosts}, \text{PSCost})
\]
6.3 Performance Requirements

This section formally defines the Performance Requirements of the worksystem. Recall from Chapter 4 that:

'A performance ordering is a combination of benefit orderings and cost orderings. The desired performance of a formal language worksystem is a combination of the desired benefits with the desired costs.'

Thus, the performance requirements are a combination of the benefit requirements and cost requirements, together with two additional components, the desired performances and the performance pre-ordering.

Any desired performance will exhibit a desired benefit while incurring desired costs. However, combinations of a desired benefit and desired cost, where the benefit which ranks low in the benefit ordering, and the cost structure ranks low in the cost ordering, might not constitute desired performances. Therefore, the desired performances are a subset of the cross-product of the desired benefits and desired costs.

\[
\text{PerformanceRequirements[WorkStructures,WorksystemCosts]} \quad \text{BenefitRequirements[WorkStructures]} \quad \text{CostRequirements[WorksystemCosts]} \\
\text{desiredPerformances} : P(\text{desiredBenefits} \times \text{desiredCosts}) \\
\text{performance} : \text{PreOrdering[domain} \times \text{possibleCosts]} \\
\text{benefit} \times \text{cost} \subseteq \text{performance} \\
\forall t,t' : \text{domain; } c,c' : \text{possibleCosts} \quad \text{•} \\
(t,c) \in \text{desiredPerformances} \\
\wedge \\
(t,c) \leq_{\text{performance}} (t',c') \\
\Rightarrow \\
(t',c') \in \text{desiredPerformances}
\]
The performance ordering operates over the combinations of work transformations and cost structures. Any performance ordering must be consistent with the benefit and cost orderings, in the sense, that any order relationship that arises from the combination of the benefit and cost orderings must be a relationship in the performance ordering. However, additional relationships might arise due to the combination. Thus, the first condition in the second box states that the product of the benefit and cost orderings is a sub-set of the performance ordering.

The second condition states that any performance which is equivalent to or improves upon a desired performance is also a desired performance.

In order to develop performance requirements for the paint shop example, it is necessary to define a set corresponding to each of the new components above. The desired performances component is addressed first.

In the paint shop, any combination of benefit and cost is a desired performance, except that any worksystem that produces only green bricks at a cost of 4 cost units is not of interest. Thus, the paint shop desired performances can be specified formally as follows:

\[
\text{PSDesiredPerformance} == \{ t : \text{PSDesiredBenefits}; c : \text{PSDesiredCosts} \mid \neg(\text{ran}(t) = \{\text{green}\} \land c = 4)\}
\]

Thus, the combination of work transformation and cost structure (TG,4) does not exhibit desired performance, even though TG exhibits desired benefit and 4 is a desired cost.
The paint shop performance ordering is a pre-ordering of the cross-product of the PSWorkDomain and the PSWorkStructures. It is defined in the following axiomatic definition:

\[ \text{PSPerformance} : \text{PreOrdering}\{\text{PSDomain} \times \text{PSPossibleCosts}\} \]

\[ \text{PSBenefits} \times \text{PSCosts} \subseteq \text{PSPerformance} \]

\[ \forall t,t' : \text{PSDomain} ; c,c' : \text{PSPossibleCosts} \\
\qquad ( (t,c) \not\in \text{PSDesiredPerformance} \\
\qquad \wedge \\
\qquad (t',c') \not\in \text{PSDesiredPerformance} ) \\
\Rightarrow \\
\qquad (t,c) =_{\text{PSPerformance}} (t',c') \]

The conditions state that:

- The product of the paint shop benefit and cost orderings is a sub-set of the paint shop performance ordering.

- Any combinations of work transformations and cost structures failing to exhibit desired performance are equal in the performance ordering.

Since \((TG,4)\) and \((TB,5)\) both fail to exhibit desired performance. The relationship

\[(TG,4) \leq_{\text{PSPerformance}} (TB,5)\]

holds, but the relationship

\[(TG,4) \leq_{(\text{PSBenefits} \times \text{PSCosts})} (TB,5)\]

does not, thus illustrating a relationship that has arisen in the performance ordering that is not part of the product of the benefit and cost orderings.
Combining all the components defined, it is possible to develop a definition for paint shop performance requirements:

\[
\text{PSPerformanceRequirements} == \\
( \text{PSDomain}, \text{PSDesiredBenefits}, \text{PSBenefit}, \text{PSPossibleCosts}, \text{PSDesiredCosts}, \text{PSCost}, \text{PSDesiredPerformance}, \text{PSPerformance})
\]

6.4 Conclusion

The conclusion to this chapter summarises the chapter and its achievements and assesses these achievements by the criteria established in the introduction (Chapter 1).

6.4.1 Summary

This chapter has developed the aspects of the formal framework concerned with specifying the desired performance of language systems using Z and the categorial extensions developed in Chapter 5. The concepts of benefit and cost requirements have been formally defined. These concepts have been combined to formally define the concept of performance requirements. A simple paint shop example has been introduced in order to provide a simple illustration of the concepts being defined.

Given any formal statement of performance requirements, including the paint shop, it is possible to formally verify whether it is a formal statement of performance requirements according to the formal framework. For the paint shop example, such a verification would amount to a formal proof of the statement:

\[
\text{PSPerformanceRequirements} \subseteq \\
\text{PerformanceRequirements}[\text{PSWorkStructures}, \text{PSWorksystemCosts}]
\]
A formal proof of the same statement would also demonstrate formally the consistency of the aspects of the formal framework concerned with performance requirements, where consistency is used here to mean that the formal definitions presented do not lead to a contradiction. In other words, there is at least one instance, the paint shop example, of the performance requirements class of the formal framework. Such a formal verification is not presented, since it is felt that the informal demonstration provided throughout the chapter will suffice.

A key question concerning any formal statement of performance requirements is whether they are an accurate reflection of the client performance requirements. Unless the client is capable of interpreting and employing the particular formal language being used for the purpose of specifying requirements, this issue can only be addressed using empirical validation techniques.

Since the paint shop example does not purport to be an example of formal language design that might interest some potential client, it remains to be demonstrated that the performance requirements aspects of the framework are appropriate for potential clients problems. To this end, Chapter 7 develops a specification of the benefit requirements for a formal language worksystem for Air Traffic Control.

6.4.2 Assessment

Recall that Chapter 1 outlined the following criteria for assessing the thesis that are applicable to the framework aspects of the thesis:

- Internal consistency between conception and framework
- External synthesis at the level of the framework
- Knowledge incremented at the level of the framework

In this chapter, the conceptual definitions established in Chapter 4 have formed the basis for the formal definitions, thus demonstrating internal consistency between the conception and framework.

Since the thesis is concerned only with the formal aspects of the framework outlined in the conception, this section is interested in the form of the
performance requirements specification rather than the process of obtaining it. However, recent work in requirements elicitation for software engineering is employed to demonstrate the external synthesis at the level of the framework and the knowledge incremented at the level of the framework.

Requirements elicitation is an important field of software engineering, as well as other disciplines. According to [Goguen 93b]:

"An enormous amount of work is relevant to requirements capture, spanning disciplines that range from philosophy and mathematics to systems engineering and certainly including sociology, linguistics, computer science, economics, business, psychology and law."

A classification of requirements elicitation techniques is reported in [Bickerton & Siddiqi 93]. Following the work of [Lytard 84] in classifying social theories, the paper develops a classification scheme for requirements elicitation methods. The scheme distinguishes between modern and post-modern methods, and within the modern methods between unitary and pluralistic approaches. Unitary approaches assume there is a pre-existing 'real system' to be 'captured'. Pluralistic approaches assume that the requirements are embodied in the sub-organisations of the organisation. For example, the different requirements of the system procurer and the system user. The modern approaches to requirements elicitation can be characterised by the belief that they have access to 'objective truth'. According to [Bickerton & Siddiqi 93]:

"For requirements engineers, post-modernism seeks to break down the certainties of modernism. In particular it attacks the belief that scientific approaches to requirements engineering [sic] have access to objective truth."

According to Goguen:

"There is very little work that falls within the post-modernist classification." [Goguen 93b]

The approach to performance requirements outlined in Chapter 4 and formalised in this chapter assumes no access to objective truth. Performance
requirements are represented as a set of desired performances and a pre-ordering of the performances. The mathematical notion of a pre-ordering is a minimal notion of an ordering enabling the consideration of incomparable systems, as well as performance equivalent but otherwise distinct systems. The notion of a set of desired performances is used because there is no optimal 'real system' to be built, even by reconciling the different goals of the sub-organisations that make up the client, but rather a (possibly empty) set of systems that are acceptable to the different sub-organisations of the client. The pre-ordering enables performances to be ordered where appropriate.

In this chapter it is the performance requirements of formal language worksystems that has been considered, not the performance requirements of software systems. Due to this difference and the large amount of work relevant to requirements capture the level of synthesis at the level of the framework is limited. However, it is possible to situate the approach, using the classification outlined above. The work on performance requirements, presented in this chapter, specifies a form for the product of post-modernist requirements elicitation in the particular case of formal language worksystems.

The approach to expressing requirements using pre-orderings, has been developed completely independently. As far as is known there has only been one similar attempt at expressing requirements in this fashion which was in the field of Architectural Planning [Ho 82]. The employment of pre-orderings and the products of pre-orderings thus marks an increment of knowledge at the level of the framework.
Chapter 6 developed the aspects of the formal framework concerned with specifying the desired performance of language worksystems. The paint shop example was introduced in order to provide a simple illustration of the concepts being defined. However, the paint shop did not purport to be a example of formal language design that might interest a potential client and, therefore, it remains to be demonstrated that the performance requirements aspects of the framework are appropriate for the problems that arise with Client Requirements. To this end, this chapter develops a specification of the benefit requirements for a formal language worksystem for Air Traffic Control (ATC). The $Z$ notation is employed to give a specification of benefit requirements consistent with the concept as defined in Chapter 6.

The first section describes some mathematical pre-requisites necessary to describe the continuous movement of aircraft through an air traffic control sector. The second section is devoted to developing the specification of the work structures, ATCWorkStructures. The third section specifies the domain and desired benefits of ATC. The final section describes the benefit ordering for ATC.

By its very nature, any statement of required benefits cannot be formally verified. The benefit requirements here are a simplification of benefit requirements of ATC. They are based upon the work of Field [Field 85] and the work of colleagues [Dowell 92]. The specification was first constructed as a basis for the development of an ATC simulator that could assess the benefits of different work systems. The simulator that was constructed, based upon the specification, provides a means of empirically validating the specification.

The original work was carried out as part of the 'Task Oriented Modelling' project sponsored by the DTI and SERC (research grant number IDE/1/171). The specification first appeared in a different form [Dowell, Salter & Long 92].
7.1 Mathematical Preliminaries

The Z notation does not contain the concept of real numbers which is the standard basis for modelling movement in three dimensional space. This section begins with an attempt to provide a specification for real numbers. The real numbers are then used to specify the points in two- and three-dimensional space. The one-dimensional interval, and the cuboid and cylinder sub-spaces of three dimensional space are then specified. Finally, the notion of differentiability and differentiation is described for functions from real intervals to one and three dimensions.

This section is presented to ensure the completeness of the specifications that follow. The reader is advised to read it lightly and to refer back to it from other sections when required. Further explanations of the specifications presented here can be found in the mathematical literature [Apostol 82], [Kreyszig 79] and [Spivak 67].

The basic type notation is used to introduce the concept of real numbers:

[Real]
The algebraic properties of the real numbers are specified using the following axiomatic description:

\[
\begin{align*}
\_+\_ &: \text{Real} \times \text{Real} \to \text{Real} \\
\_\ast\_ &: \text{Real} \times \text{Real} \to \text{Real} \\
\_\_\_ &: \text{Real} \to \text{Real} \\
\_\_\_^{-1} &: \text{Real} \to \text{Real}
\end{align*}
\]

\[\forall x : \mathbb{Z} \cdot i \in \text{Real}\]

\[\forall a, b, c : \text{Real} \cdot \]

\[\begin{align*}
a + (b + c) &= (a + b) + c \\
\land a + 0 &= a \\
\land 0 + a &= a \\
\land a + (-a) &= 0 \\
\land (-a) + a &= 0 \\
\land a + b &= b + a \\
\land a \ast (b \ast c) &= (a \ast b) \ast c \\
\land \neg (1 = 0) \\
\land a \ast 1 &= a \\
\land 1 \ast a &= a \\
\land \neg (a = 0) \Rightarrow a \ast a^{-1} &= 1 \\
\land \neg (a = 0) \Rightarrow a^{-1} \ast a &= 1 \\
\land a \ast b &= b \ast a \\
\land a \ast (b + c) &= a \ast b + b \ast c
\end{align*}\]
The ordering properties of the real numbers are specified using the following axiomatic description:

<table>
<thead>
<tr>
<th>PositiveReals</th>
<th>: \mathbb{P} \text{Real}</th>
</tr>
</thead>
<tbody>
<tr>
<td>NegativeReals</td>
<td>: \mathbb{P} \text{Real}</td>
</tr>
<tr>
<td>&lt;</td>
<td>: \text{Real} \leftrightarrow \text{Real}</td>
</tr>
<tr>
<td>&gt;</td>
<td>: \text{Real} \leftrightarrow \text{Real}</td>
</tr>
<tr>
<td>\leq</td>
<td>: \text{Real} \leftrightarrow \text{Real}</td>
</tr>
<tr>
<td>\geq</td>
<td>: \text{Real} \leftrightarrow \text{Real}</td>
</tr>
</tbody>
</table>

\forall a,b : \text{Real} \bullet
\quad a \in \text{PositiveReals} \iff (-a) \in \text{NegativeReals}
\land \quad \cup \{\text{NegativeReals},\{0\},\text{PositiveReals}\} = \text{Real}
\land \quad \cap \{\text{NegativeReals},\{0\},\text{PositiveReals}\} = \emptyset

\forall a,b : \text{PositiveReals} \bullet
\quad a + b \in \text{positiveReals}
\land \quad a \times b \in \text{positiveReals}

\forall a,b : \text{Real} \bullet
\quad a > b \iff a - b \in \text{PositiveReals}
\land \quad a < b \iff b > a
\land \quad a \geq b \iff (a > b \lor a = b)
\land \quad a \leq b \iff (a < b \lor a = b)
The least upper bound properties of the real numbers are specified using the following axiomatic description:

\[
\begin{align*}
\text{isAnUpperBoundOf} & : \text{Real} \leftrightarrow \mathcal{P}\text{Real} \\
\text{leastUpperBound} & : \mathcal{P}\text{Real} \to \text{Real}
\end{align*}
\]

\[
\forall x : \text{Real} ; A : \mathcal{P}\text{Real} \bullet \\
\quad \text{x isAnUpperBoundOf A} \iff \forall a:A \bullet x \geq a \\
\quad \land \text{leastUpperBound(A) isAnUpperBoundOf A} \\
\quad \land \text{x isAnUpperBoundOf A} \\
\quad \Rightarrow x \geq \text{leastUpperBound(A)}
\]

\[
\text{dom(leastUpperBound)} = \\
\{ A : \mathcal{P}\text{Real} \mid \\
\quad A \neq \emptyset \\
\quad \land \exists x : \text{Real} \bullet x \text{ isAnUpperBoundOf A} \}
\]

The additional operations of absolute value, square, subtraction and division are specified for real numbers using the following axiomatic description:

\[
\begin{align*}
\text{|-|} & : \text{Real} \to \text{Real} \\
\text{\_\_2} & : \text{Real} \to \text{Real} \\
\text{\_\_-} & : \text{Real} \times \text{Real} \to \text{Real} \\
\text{\_\_/} & : \text{Real} \times \text{Real} \to \text{Real}
\end{align*}
\]

\[
\forall a,b : \text{Real} \bullet \\
\quad (a \geq 0 \Rightarrow |a| = a) \\
\quad \land (a \leq 0 \Rightarrow |a| = -a) \\
\quad \land a^2 = a \cdot a \\
\quad \land a - b = a + (-b) \\
\quad \land a/b = a \cdot (b^{-1})
\]
The concept of a sum of real numbers is specified using a generic axiomatic description. For present purposes, it will be convenient to sum the set of real numbers that form the range of a partial function:

\[
\begin{align*}
\text{sum} & : (X \to \text{Real}) \to \text{Real} \\
\text{sum}(\emptyset) & = 0 \\
\forall f : X \to \text{Real} \bullet \\
\forall x : \text{dom}(f) \bullet \\
\text{sum}(f) & = \text{sum}(\{x\} \lhd f) + f(x)
\end{align*}
\]

The conditions state:

- The sum of a function with no elements in the domain is zero.
- For any function and element, belonging to the domain of that function, the sum of the function is equal to the sum of the function with the element removed from the domain, plus the value of the function for that element.

So for example:

\[
\text{sum}(\{(a,2), (b,2), (c,5)\}) = 9
\]

The set of points in two-dimensional real space is specified using the Z schema notation:

\[
\begin{align*}
\text{Real2} & \\
\text{latitude} & : \text{Real} \\
\text{longitude} & : \text{Real}
\end{align*}
\]

A point in two-dimensional real space consists of two real numbers, the latitude and the longitude. There are no conditions.
The set of points in three-dimensional real space is specified using the Z schema notation:

\[ \text{Real3}_z \]

\[ \text{Real2} \]
\[ \text{altitude} : \text{Real} \]

A point in three-dimensional real space consists of a point in two-dimensional real space together with an additional real number representing altitude. There are no conditions.

The operations of scalar product, norm, subtraction and lying between are specified for three-dimensional points in the following axiomatic description:

\[ \forall a, b, c : \text{Real3} \cdot \]
\[ a \cdot b = a.\text{latitude} \cdot b.\text{latitude} \]
\[ + a.\text{longitude} \cdot b.\text{longitude} \]
\[ + a.\text{altitude} \cdot b.\text{altitude} \]
\[ |a|^2 = a \cdot a \]
\[ a \cdot b = (a.\text{latitude} - b.\text{latitude}, a.\text{longitude} - b.\text{longitude}, a.\text{altitude} - b.\text{altitude}) \]
\[ (b \text{ liesBetween a and c} \iff |a-b| + |b-c| = |a-c|) \]
The set of real intervals is specified using the Z schema notation:

```
RealInterval

min : Real
max : Real
size : Real
values : P Real
```

A real interval consists of three real numbers: min, max and the size, together with a set of real numbers called values.

The conditions state that:

- The min is always less than the max.
- The size is the difference between the minimum and the maximum.
- Values is the set of all numbers between min and max inclusive.

Figure 7.1 illustrates the attributes of a real interval.
The set of three-dimensional cuboids is specified using the schema notation:

```
<table>
<thead>
<tr>
<th>Cuboid</th>
</tr>
</thead>
<tbody>
<tr>
<td>latitudeRange : RealInterval</td>
</tr>
<tr>
<td>longitudeRange : RealInterval</td>
</tr>
<tr>
<td>altitudeRange : RealInterval</td>
</tr>
<tr>
<td>values : P Real3</td>
</tr>
</tbody>
</table>
```

\[
\text{values} = \text{latitudeRange.values} \times \text{longitudeRange.values} \times \text{altitudeRange.values}
\]

A cuboid consists of three real intervals: the latitude, longitude and altitude ranges, as well as a set of three-dimensional points which represent the values of the cuboid. The condition states that the set of values of the cuboid is equal to the cross-product of the values of each of the ranges.

A cuboid is represented in the following Figure 7.2.
The set of all three-dimensional cylinders is specified:

<table>
<thead>
<tr>
<th>Cylinder</th>
</tr>
</thead>
<tbody>
<tr>
<td>centre   : Real2</td>
</tr>
<tr>
<td>radius   : PositiveReals</td>
</tr>
<tr>
<td>altitudeRange : RealInterval</td>
</tr>
<tr>
<td>values   : ( \mathbb{P} ) Real3</td>
</tr>
</tbody>
</table>

\[
\forall v : \text{Real3} \cdot \\
\quad v \in \text{values} \\
\quad \iff \\
\quad \left( \left( (v.\text{latitude} - \text{centre.}\text{latitude})^2 + (v.\text{longitude} - \text{centre.}\text{longitude})^2 \leq \text{radius}^2 \right) \land \\
\quad \quad v.\text{altitude} \in \text{altitudeRange.}\text{values} \right)
\]

A cylinder consists of a two-dimensional point representing the centre, a positive real representing the radius, a real interval representing the altitude range and, finally, a set of three-dimensional points representing the set of values that make up the cylinder.

The condition states that a value is one of the values of the cylinder if and only if, the latitude and longitude of the value are within radius distance of the centre and the altitude is within the altitude range. A cylinder is illustrated in Figure 7.3.
Figure 7.3 The Attributes of the Cylinder
The concept of differentiation from the reals to the reals is specified using the following axiomatic description:

\[ \text{RealDifferentiableOn : } \text{RealInterval} \rightarrow \mathcal{P} \text{(Real } \rightarrow \text{ Real)} \]

\[ \text{' : } \mathcal{P} \text{(Real } \rightarrow \text{ Real) } \rightarrow \text{ (Real } \rightarrow \text{ Real)} \]

For all \( f : \text{Real} \rightarrow \text{Real} \), \( a,d : \text{Real} \):

\[ (a \in \text{dom}(f') \land f'(a) = d) \]

\[ \equiv \]

\[ \forall \varepsilon : \text{PositiveReals} \cdot \exists \delta : \text{PositiveReals} \cdot \forall x : \text{Real} : \]

\[ (|x - a| > 0 \land |x - a| < \delta) \]

\[ \Rightarrow \]

\[ |(f(x+a) - f(a))/x - d| < \varepsilon \]

For all \( \text{interval} : \text{RealInterval} \)

\[ \text{RealDifferentiableOn interval} = \]

\[ \{ f : \text{interval.values } \rightarrow \text{Real} \mid \text{dom}(f') = \text{interval.values} \} \]

In order to describe what we mean by the differential of a function from the reals to the set of three-dimensional points, \( \text{Real}^3 \), we need to specify the latitude, longitude and altitude component functions. This specification is achieved in the following axiomatic description:

\[ \text{latitude} : (\text{Real} \rightarrow \text{Real}^3) \rightarrow (\text{Real} \rightarrow \text{Real}) \]

\[ \text{longitude} : (\text{Real} \rightarrow \text{Real}^3) \rightarrow (\text{Real} \rightarrow \text{Real}) \]

\[ \text{altitude} : (\text{Real} \rightarrow \text{Real}^3) \rightarrow (\text{Real} \rightarrow \text{Real}) \]

For all \( f : \text{Real} \rightarrow \text{Real}^3 \), \( a : \text{Real} \):

\[ \text{latitude}(f)(a) = f(a).\text{latitude} \]

\[ \land \]

\[ \text{longitude}(f)(a) = f(a).\text{longitude} \]

\[ \land \]

\[ \text{altitude}(f)(a) = f(a).\text{altitude} \]
The concept of differentiation from the reals to three-dimensional points is specified using the following axiomatic description:

\[
\text{Real3DifferentiableOn} : \text{RealInterval} \rightarrow \mathcal{P}(\text{Real} \leftrightarrow \text{Real3})
\]

\[
_· : \mathcal{P}(\text{Real} \leftrightarrow \text{Real3}) \leftrightarrow (\text{Real} \leftrightarrow \text{Real3})
\]

\[
\forall f : \text{Real} \leftrightarrow \text{Real3}; a : \text{Real} \bullet f'(a) = (\text{latitude}(f)'(a), \text{longitude}(f)'(a), \text{altitude}(f)'(a))
\]

\[
\forall \text{interval : RealInterval}
\text{Real3DifferentiableOn interval} =
\{ f : \text{interval.values} \rightarrow \text{Real3} \mid \text{dom}(f') = \text{interval.values} \}
\]

This section aimed to specify the notions of a cuboid and cylinder so as to aid the specification of air traffic control work structures. It also aimed to define the notions of differentiability for functions from real intervals to one and three dimensions. In order to achieve these objectives, the set of points for two- and three-dimensional space were specified. These specifications in turn were built upon the specification of the real numbers.
7.2 Work Structures

In this section, the work structures of air traffic control are described. The section begins by describing the concepts of aircraft, beacons, planned and actual flights. A class of air traffic control sectors is defined and extended with the inclusion of a set of planned flights to a class of air traffic control scenarios. Finally, an air traffic control work structure is defined as a class of particular actualisations of air traffic scenarios.

The specification dependency structure of the specifications which are used to construct the air traffic control work structures are illustrated in Figure 7.4. In this figure, a link occurs between two specifications, if the lower one was directly used in the construction of the upper one.

The first specification defines limits on aircraft climbing ability:

\[
\text{climbingAbilityMax} : \mathbb{N}
\]

A class of beacons is specified:

\[
\text{ATCBeacon} : \text{Cylinder}
\text{handoverRanges} : \mathcal{F} \text{ RealInterval}
\]

\[
\forall i : \text{handoverRanges} \cdot i \subseteq \text{position.altitudeRange}
\]

A beacon consists of a cylinder called the position, together with a finite set of one-dimensional intervals called the hand-over ranges. The hand-over ranges are all contained within the altitude range of the position.

The attributes of a beacon are illustrated in the Figure 7.5.
Figure 7.4 The Dependencies Between Components in the Specification of ATC Work Structures
Figure 7.5 The Attributes of a Beacon
A class of air traffic control aircraft are defined:

```
ATCAircraft

possibleSpeeds : RealInterval
cruisingSpeed : Real
climbingAbility : 1..climbingAbilityMax
accelerationAbility : PositiveReals
fuelConsumptionRate : RealDifferentiableOn possibleSpeeds
```

- `possibleSpeeds.min ≥ 0`
- `cruisingSpeed ∈ possibleSpeeds.values`
- `∀s : possibleSpeeds.values • fuelConsumptionRate(s) > 0`
- `∀s : possibleSpeeds.values • fuelConsumptionRate'(cruisingSpeed) = 0`
- `(s < cruisingSpeed ⇒ fuelConsumptionRate'(s) < 0)`
- `(s > cruisingSpeed ⇒ fuelConsumptionRate'(s) > 0)`

Air traffic control aircraft consist of five components: a real interval representing possible speeds; a positive real number representing cruising speed; a climbing ability which ranges from one to the maximum climbing ability; a real number representing acceleration ability; and, finally, a real differentiable function on possible speeds representing the fuel consumption rate. The fuel consumption rate represents the rate of change of fuel consumption with respect to speed.

The conditions state that:

- The minimum possible speed is greater than or equal to zero.
- The cruising speed is within the possible speeds.
- Fuel consumption is greater than zero at all speeds.
- Fuel consumption is at a minimum at cruising speed.

Aircraft turning ability is not mentioned in the specification as it is assumed that aircraft can turn instantaneously.
The relationship between fuel consumption rate, speed and cruising speed is illustrated in Figure 7.6.

Figure 7.6 The Relationship Between Fuel Consumption Rate, Speed and Cruising Speed
An air traffic control flight plan is an extension of the ATCAircraft schema:

\[
\text{ATCFlightPlan} \quad \text{ATCAircraft}
\]

\[
\begin{align*}
\text{route} & : \text{iseq ATCBeacon} \\
\text{firstBeacon} & : \text{ATCBeacon} \\
\text{lastBeacon} & : \text{ATCBeacon} \\
\text{arrivalPosition} & : \text{Real3} \\
\text{arrivalTime} & : \text{Real} \\
\text{arrivalSpeed} & : \text{Real} \\
\text{departureRange} & : \text{RealInterval}
\end{align*}
\]

\[
\begin{align*}
\text{firstBeacon} & = \text{head}(	ext{route}) \\
\text{lastBeacon} & = \text{last}(	ext{route}) \\
\text{arrivalPosition.latitude} & = \text{firstBeacon.position.centre.latitude} \\
\text{arrivalPosition.longitude} & = \text{firstBeacon.position.centre.longitude} \\
\text{arrivalSpeed} & \in \text{possibleSpeeds.values} \\
\text{departureRange} & \in \text{lastBeacon.handoverRanges}
\end{align*}
\]

In addition to the aircraft attributes, a flight plan consists of: an injective sequence\(^1\) of beacons called the route; two beacons, called the first beacon and the last beacon; a three dimensional point representing the arrival position; two real numbers which represents the planned arrival time and arrival speed; and real interval called the departure range.

The conditions state:

- The first beacon is the head of the route sequence.
- The last beacon is the last member of the route sequence.
- The plane arrives at the centre of the beacon.
- The arrival speed is contained within the possible speeds of the aircraft.
- The departure range is a hand-over range of the last beacon.

---

\(^1\) An injective sequence is a sequence in which each value only occurs once.
A schema is used to define the concept of an air traffic control sector:

\[
\text{ATCSector}
\]

- **sector** : Cuboid
- **beacons** : \(\mathcal{F}\) ATCBeacon
- **legalRoutes** : beacons \(\leftrightarrow\) beacons
- **corridorSpace** : \((\text{beacons} \times \text{beacons}) \rightarrow \mathcal{P}\) Real3
- **legalSectorSpace** : \(\mathcal{P}\) Real3
- **minVertSep** : PositiveReals
- **minTimeSep** : PositiveReals
- **speedLimit** : PositiveReals

\[
\forall \text{beacon}\colon \text{beacons} \smallbullet \\
\text{sector.values} \supseteq \text{beacon.position.values}
\]

\[
\forall \text{beacon}\colon \text{beacons} \smallbullet \\
\text{beacon.position.altitudeRange} = \text{sector.altitudeRange}
\]

\[
\forall \text{beacon}_1,\text{beacon}_2\colon \text{beacons} \smallbullet \\
\text{beacon}_1.position.values \cap \text{beacon}_2.position.values = \emptyset
\]

\[
\forall \text{beacon}_1,\text{beacon}_2\colon \text{beacons} \smallbullet \\
(\text{beacon}_1,\text{beacon}_2) \in \text{legalRoutes}^* \\
\Rightarrow \\
(\text{beacon}_1,\text{beacon}_2) \in \text{legalRoutes} \\
\land \text{point} \notin \text{beacon}_1.values \cup \text{beacon}_2.values \\
\land \exists \text{p}_1\colon \text{beacon}_1.values; \text{p}_2; \text{beacon}_2.values \smallbullet \\
\text{point liesBetween p}_1 \text{ and p}_2)
\]

\[
\text{legalSectorSpace} = \\
(\cup \{\text{Points}\colon \mathcal{P}\) Real3 \mid \exists \text{beacons} \smallbullet \text{Points} = \text{b.values}\}) \\
\cup \ (\cup \ \text{ran}(\text{corridorSpace}))
\]

\[
\forall \text{beacon}\colon \text{beacons} \smallbullet \\
\text{beacon.position.radius} \geq \text{minTimeSep}^*\text{speedLimit}
\]
The air traffic control sector consists of a cuboid called the sector which represents: the three-dimensional space of the sector; a finite sub set of the set ATCBeacon, called the beacons; a relation upon the beacons called the legal routes; a function from pairs of beacons to sets of three-dimensional points called the corridorSpace; a set of three-dimensional points called the legal sector space; together with three real numbers representing the minimum vertical and time separation and the sector speed limit.

The conditions state:

- All the beacons are contained in the sector.
- The altitude range of the position of the beacon is the same as the altitude range of the sector.
- No beacons occupy the same space.
- The nature of the legal routes relation is such that it is possible to fly from any beacon to any other beacon.
- The corridor space is the set of all points which lie between two beacons which are on a legal route.
- The legal sector space is the union of the space occupied by the beacons and the space occupied by the corridors.
- The radius of the beacons must be big enough to contain aircraft flying at the speed limit and the minimum time separation.

The attributes of sector, beacons and corridors are illustrated in Figure 7.7 and legal sector space of the same sector is illustrated in Figure 7.8.

A partial function, safe sector throughput, is defined from the set of all possible air traffic control sectors to the natural numbers.

\[
safeSectorThroughput : ATCSector \rightarrow \text{Real}
\]

Safe sector throughput represents the limit placed upon certain sectors by the air traffic authority.
Figure 7.7 The Sector, Beacon and Corridor Attributes of an ATC Sector
Figure 7.8 The Legal Sector Space for the ATC Sector of Figure 7.7
The ATCSector schema is extended in the definition of an ATCScenario:

---

<table>
<thead>
<tr>
<th>ATCScenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATCSector</td>
</tr>
<tr>
<td>controlTimePeriod : RealInterval</td>
</tr>
<tr>
<td>plannedFlights : ( \mathcal{F} ) ATCFlightPlan</td>
</tr>
<tr>
<td>throughput : Real</td>
</tr>
</tbody>
</table>

\[
\forall \text{flight} : \text{plannedFlights} \implies \\
\text{flight.arrivalTime} \leq \text{controlTimePeriod.max} \\
\land \text{beacons} \supseteq \text{ran(flight.route)} \\
\land \forall n:1..\text{flight.route}^{#-1} \implies \\
(\text{flight.route}(n),\text{flight.route}(n+1)) \in \text{legalRoutes} \\
\land \text{flight.arrivalPosition} + \text{minVertSep} \leq \text{sector.altitudeRange.max} \\
\land \text{flight.arrivalPosition} - \text{minVertSep} \geq \text{sector.altitudeRange.min} \\
\land \text{flight.arrivalSpeed} \leq \text{speedLimit} \\
\]

\[
\text{throughput} = \frac{\text{plannedFlights}^\#}{\text{controlTimePeriod.size}}
\]

An air traffic control scenario is an air traffic control sector, together with a real interval which represents the control time period and a finite set of flight plans, called planned flights.

The first condition states that for every planned flight:

- The arrival time is before the end of the control time period.
- The route is made up of sector beacons.
- Every adjacent pair of beacons in the flight route is a legal route of the sector.
- The arrival altitude is safely within the sector.
- The arrival speed is less than the speed limit.

The second condition states that throughput is equal to the number of planned flights divided by the control time period.
Two schemas are used to define the concept of an actual flight. In the first, ATCAircraft schema is extended to define an ATCBasicActualFlight Schema:

---

**ATCBasicActualFlight**

ATCAircraft

- minTimeSep : PositiveReals
- minVertSep : PositiveReals
- timePeriod : RealInterval
- speed : RealDifferentiableOn timePeriod
- position : Real3DifferentiableOn timePeriod
- fuelConsumption : RealDifferentiableOn timePeriod
- separationSpace : timePeriod.values → Cylinder

∀t:timePeriod.values •

- speed(t) ∈ possibleSpeeds.values
- altitude(position)'(t)/speed(t) ≤ climbingAbility/climbingAbilityMax
- speed'(t) ≤ accelerationAbility
- speed(t) = |position'(t)|
- fuelConsumption'(t) = fuelConsumptionRate(speed(t))
- ∀ c : Cylinder •
  separationSpace(t) = c
  ⇒ (c.center = (position(t).latitude, position(t).longitude))
  ∧
  c.radius = minTimeSep*speed(t)
  ∧
  c.altitudeRange.min = position(t).altitude - minVertSep
  ∧
  c.altitudeRange.max = position(t).altitude + minVertSep
---

198
In addition to the aircraft attributes, an air traffic control basic actual flight consists of a real interval representing the time period of the flight, a real differentiable function on the time period representing speed, a three-dimensional real differentiable function on the time period representing position, a real differentiable function on the time period representing fuel consumption, two positive real numbers representing minimum time and vertical separation, and a function from time period values to cylinders representing the separation space occupied by the plane.

The conditions state that at any time during the time period:

- The speed function is within the possible speed values.
- The rate of change of altitude is constrained with respect to the speed in the ratio of climbingAbility : maxClimbingAbility.
- The rate of change of speed is less than acceleration ability.
- Aircraft speed is equal to the rate of change of distance.
- The rate of change of fuel consumption is equal to the fuel consumption rate at the given speed.
- The separation space is a cylinder with the position of the aircraft in the centre, whose height is twice the minimum vertical separation and whose radius is the product of speed and minimum time separation.

Typical position, speed, fuel consumption and separation space functions are illustrated in Figures 7.9 - 7.11 below.

Figure 7.9 An Example of a Position Attribute of a Basic Actual Flight.
Figure 7.10 Examples of a Speed and Fuel Consumption Attributes of a Basic Actual Flight.

Figure 7.11 An Example of a Separation Space Attribute of a Basic Actual Flight over a Time Period.
In the second schema the air traffic control actual flight is specified:

\[
\text{\texttt{ATCAActualFlight}}
\]

\[
\text{\texttt{ATCBasicActualFlight}}
\]

\[
\text{possiblePredictions : } \mathcal{P} \text{ \texttt{ATCBasicActualFlight}}
\]

\[
\forall \text{ prediction : possiblePredictions} \cdot \\
\quad \text{prediction\_possibleSpeeds = possibleSpeeds} \\
\quad \text{prediction\_cruisingSpeed = cruisingSpeed} \\
\quad \text{prediction\_climbingAbility = climbingAbility} \\
\quad \text{prediction\_accelerationAbility = accelerationAbility} \\
\quad \text{prediction\_fuelConsumptionRate = fuelConsumptionRate} \\
\quad \text{prediction\_minTimeSep = minTimeSep} \\
\quad \text{prediction\_minVertSep = minVertSep}
\]

\[
\forall \text{ prediction : possiblePredictions'} \cdot \\
\quad \text{prediction\_timePeriod\_min = timePeriod\_max} \\
\quad \text{prediction\_position(prediction\_timePeriod\_min) = position(timePeriod\_max)} \\
\quad \text{prediction\_fuelConsumption(prediction\_timePeriod\_min) = fuelConsumption(timePeriod\_max)} \\
\quad \forall t: \text{ prediction\_timePeriod\_values} \cdot \\
\quad \text{prediction\_speed(t) = speed(timePeriod\_max)}
\]

An air traffic control actual flight consists of an air traffic control basic actual flight together with a set of air traffic control basic flights which represents the possible predictions of the flight after the flight's time period.

The first condition states that for any possible prediction, all of the aircraft attributes and the separation constraints are the same as those for the actual flight.
The second condition states that for any possible prediction:

- The prediction time period follows immediately after the time period.
- The prediction position at the beginning of the prediction time period is the position at the end of the time period.
- The prediction fuel consumption at the beginning of the prediction period is the fuel consumption at the end of the time period.
- Throughout the prediction time period the prediction speed is the speed at the end of the time period.

The way in which possible predictions extend the actual flight of an aircraft beyond the time period is illustrated in Figure 7.12.
Figure 7.12 An Example of the Possible Predictions of an Actual Flight.
The basic type definition is used to introduce the air traffic control notion of safety:

\[ \text{ATCSafety} = \{ \text{safe, unsafe} \} \]

The benefit ordering for ATC will be dependant upon the safety attribute of the work structure. The specification of the benefit ordering can be simplified by defining a pre-ordering of ATCSafety. This specification is presented in the following axiomatic description:

\[
\text{ATCSafe} : \text{PreOrdering}[\text{ATCSafety}]
\]

\[ \text{unsafe} <_{\text{ATCSafe}} \text{safe} \]

The condition states, that for the ATC ordering, unsafe is strictly less than safe.

The other attributes of an ATC work structure that will affect the benefit ordering are the fuel consumption and the estimated flying time. As the value of both decrease, the value of the ATC benefit improves. Thus, to ease the ATC benefit specification, the following pre-ordering is defined which is the opposite of the normal real ordering:

\[
\text{Real}^{\text{OP}} : \text{PreOrdering}[\text{Real}]
\]

\[
\forall x, y : \text{Real} \\
\quad x \leq y \\
\quad \iff \\
\quad y \leq_{\text{Real}^{\text{OP}}} x
\]

The condition states that two numbers are related in the Real\(^{\text{OP}}\) ordering if they have the opposite relationship in the real ordering.
An air traffic control work structure represents the actualisation of a given air traffic control scenario. The conditions of a work structure are quite lengthy and are therefore split into eleven sections which are then connected using the schema conjunction operator. Each of the eleven schemas consists of an air traffic control scenario together with a set of attributes. These attributes are defined in the work structure attributes schema:

\[
\begin{align*}
\text{ATCScenario} & \\
\text{actual} & : \text{plannedFlights} \to \text{ATCActualFlight} \\
\text{collisions} & : \text{plannedFlights} \leftrightarrow \text{plannedFlights} \\
\text{plannedExits} & : \mathcal{P} \text{plannedFlights} \\
\text{enRoute} & : \mathcal{P} \text{plannedFlights} \\
\text{unplannedExits} & : \text{plannedFlights} \to \mathcal{P} \text{plannedFlights} \\
\text{predicted} & : \text{plannedFlights} \to \text{ATCActualFlight} \\
\text{safety} & : \text{ATCSafety} \\
\text{estimatedTotalFuelConsumption} & : \text{Real} \\
\text{estimatedTotalFlyingTime} & : \text{Real}
\end{align*}
\]

The work structure attributes consist of the scenario attributes together with: a partial function, actual, which maps planned flights to actual flights; a relation between planned flights representing collisions; three subsets of planned flights representing aircraft which by the end of the control time period, have had planned exits, are en route or have had unplanned exits; a function from the set of planned flights to the set of sets of flight predictions which represent the predicted flight of a planned flight; a value of ATCSafety representing safety; and two real numbers representing estimated total fuel consumption and flying time.
The eleven schemas are concerned with specifying relationships which determine:

- conditions which hold throughout the control time period.
- the start of the time period for each flight.
- whether aircraft remain within the legal sector space.
- collisions.
- planned exits.
- when a plane is en route at the end of the control time period.
- unplanned exits.
- the predicted period for each flight en route at the end of the control time period.
- safety.
- estimated total fuel consumption.
- estimated total flying time.
The first schema describes conditions that hold throughout the period:

\[ \text{ATCFixedConditions} \]

\[ \text{ATCWorkStructuresAttributes} \]

\[
\forall \text{flight}: \text{plannedFlights} \cdot \\
\quad \text{flight}.\text{possibleSpeeds} = \text{actual(flight)}.\text{possibleSpeeds} \\
\quad \wedge \\
\quad \text{flight}.\text{cruisingSpeed} = \text{actual(flight)}.\text{cruisingSpeed} \\
\quad \wedge \\
\quad \text{flight}.\text{climbingAbility} = \text{actual(flight)}.\text{climbingAbility} \\
\quad \wedge \\
\quad \text{flight}.\text{accelerationAbility} = \text{actual(flight)}.\text{accelerationAbility} \\
\quad \wedge \\
\quad \text{flight}.\text{fuelConsumptionRate} = \text{actual(flight)}.\text{fuelConsumptionRate} \\
\quad \wedge \\
\quad \text{controlTimePeriod}.\text{values} \supseteq \text{actual(flight)}.\text{timePeriod}.\text{values} \\
\quad \wedge \\
\quad \text{minTimeSep} = \text{actual(flight)}.\text{minTimeSep} \\
\quad \wedge \\
\quad \text{minVertSep} = \text{actual(flight)}.\text{minVertSep} \\
\quad \wedge \\
\quad \forall t : \text{actual(flight)}.\text{timePeriod}.\text{values} \cdot \\
\quad \text{actual(flight)}.\text{speed}(t) \leq \text{speedLimit} \\
\]

\[
\forall \text{flight1,flight2}: \text{plannedFlights} \cdot \\
\quad (\text{flight1,flight2} \in \text{collisions} \iff (\text{flight2,flight1}) \in \text{collisions}) \\
\]

The conditions state:

- For any flight: planned and actual, possible speeds, cruising speeds, climbing abilities, acceleration ability and fuel consumption rates are equal; the time period is contained in the control time period; the minimum time and vertical separation are equal to those of the sector; and the speed of the flight is always within the speed limit.

- The collision relation is symmetric
The next schema describes the start conditions of each flight:

\[ \text{ATCStartConditions} \]

\[ \text{ATCWorkStructuresAttributes} \]

\[ \forall \text{ flight: plannedFlights; } \]
\[ t_0: \text{Real; } \]
\[ \text{unsafeInterval : RealInterval} \cdot \]
\[ t_0 = \text{actual(flight).timePeriod.min} \]
\[ \wedge \]
\[ \text{unsafeInterval.min = flight.arrivalTime} \]
\[ \wedge \]
\[ \text{unsafeInterval.max = t_0} \]
\[ \wedge \]
\[ \forall t : \text{unsafeInterval.values \setminus \{unsafeInterval.max\}} \]
\[ \exists f : \text{plannedFlights} \cdot \]
\[ \text{actual(f).separationSpace(t).values} \]
\[ \cap \]
\[ \text{flight.firstBeacon.position.values} \neq \emptyset \]
\[ \wedge \]
\[ \text{actual(flight).position}(t_0) = \text{arrivalPosition} \]
\[ \wedge \]
\[ \text{actual(flight).speed}(t_0) = \text{flight.arrivalSpeed} \]

The condition states that for any flight, initial instant, unsafe interval:

- The initial instant is equal to the beginning of the time period of the actual flight.
- The beginning of the unsafe interval is the flight arrival time.
- The end of the unsafe interval is the initial instant.
- For all values of the unsafe interval, except the last, there is another plane contained within the flight's first beacon.
- The position of the actual flight at the initial instant is the planned arrival position.
- The speed of the actual flight at the initial instant is the planned arrival speed.
The next schema defines the inside sector condition:

\[
\text{ATCInsideSectorCondition} \\
\text{ATCWorkStructuresAttributes}
\]

\[
\forall \text{ flight : plannedFlights; } \ 
\forall \text{ t : actual(flight).timePeriod.values } \cdot \\
\text{legalSectorSpace} \supseteq \text{actual(flight).separationSpace(t).values} \\
\lor \\
\text{t = actual(flight).timePeriod.max}
\]

The condition states that for any flight, and instant during that flight, either the flight's separation space is contained in the legal sector space or the flight ends at that instant.
The next schema defines the notion of a collision:

\[
\text{ATCCollisionCondition} \quad \text{ATCWorkStructuresAttributes}
\]

\[
\forall \text{flight1,flight2: plannedFlights} ; \ t: \text{controlTimePeriod.values}\cdot \\
\quad (\text{flight1} \neq \text{flight2} \\
\quad \land \\
\quad \text{actual(flight1).separationSpace(t)} \\
\quad \cap \\
\quad \text{actual(flight2).separationSpace(t) \neq \emptyset}) \\
\implies \\
\quad (\text{actual(flight1).timePeriod.max} = t \\
\quad \land \\
\quad \text{actual(flight2).timePeriod.max} = t \\
\quad \land \\
\quad (\text{flight1,flight2} \in \text{collisions})
\]

The condition states that for any two distinct flights and any instant in time, if the separation space of the flights overlap at that instant, then the flights end at that instant in a collision.
The next schema defines the notion of a planned exit:

\[
\text{ATCPlannedExitCondition} \qquad \text{ATCWorkStructuresAttributes}
\]

\[
\forall \text{flight: plannedFlights}; t : \text{actual(flight).timePeriod.values} \cdot \\
(\text{flight.lastBeacon.position.values} \supseteq \\
\text{actual(flight).separationSpace(t).values} \wedge \\
\text{flight.departureRange} \supseteq \\
\text{actual(flight).separationSpace(t).altitudeRange} \wedge \\
\text{flight} \notin \text{dom(collisions)}) \\
\Rightarrow \\
(\text{instant} = \text{actual(flight).timePeriod.max} \wedge \\
\text{flight} \in \text{plannedExits}))
\]

The conditions state that for any flight and any instant of that flight, if the separation space of the flight is within the last beacon of the flight, the altitude range of the flight is contained within the departure range of the flight, and the plane is not involved in a collision, then the flight is completed with a planned exit.
The next schema describes the en Route condition:

\[
\forall \text{flight} : \text{plannedFlights} \bullet \\
(\text{flight} \notin \text{dom(collisions)}) \\
\wedge \\
\text{flight} \notin \text{plannedExits} \\
\wedge \\
\text{legalSectorSpace} \geq \text{actual(flight).separationSpace(t).values} \\
\wedge \\
\text{actual(flight).timePeriod.max} = \text{controlTimePeriod.max} \\
\Rightarrow \\
\text{flight} \in \text{enRoute}
\]

The condition states that a plane which has not collided or had a planned exit from the sector, whose separation space is always within the legal sector space but, whose time period ends when the control time period ends, is en route.
The next schema describes the set of planes that have had unplanned exits from the sector:

\[ \text{ATCUnplannedExitCondition} \]

\[ \text{ATCWorkStructuresAttributes} \]

\[ \forall \text{flight} : \text{plannedFlights} \cdot \\
(\text{flight} \notin \text{dom(collisions)} \\
\land \\
\text{flight} \notin \text{plannedExits} \\
\land \\
\text{flight} \notin \text{enRoute}) \\
\Rightarrow \\
\text{flight} \in \text{unplannedExits} \]

The condition states that any plane which has not collided, had a planned exit, or is still en route, belongs to the set of unplanned exits.
The next schema specifies the predicted flights of all en route planes:

\[ \text{ATCPredictedFlightCondition} \]

\[ \text{ATCWorkStructuresAttributes} \]

\[ \forall \text{flight: plannedFlights}; \]
\[ \quad \text{prediction : actual(flight).possiblePredictions} \]
\[ \quad t : \text{prediction.timePeriod.values} \]
\[ \quad \text{legalSectorSpace} \supseteq \text{prediction.separationSpace(t).values} \]
\[ \quad \land \]
\[ \quad \text{flight.lastBeacon.position.values} \supseteq \]
\[ \quad \text{prediction.separationSpace(prediction.timePeriod.max).values} \]
\[ \quad \land \]
\[ \quad \text{flight.lastBeacon.position.altitudeRange} \supseteq \]
\[ \quad \text{prediction.separationSpace(prediction.timePeriod.max).altitudeRange} \]

\[ \forall \text{flight: plannedFlights} \]
\[ \quad \text{predicted(flight) } \epsilon \text{ actual(flight).possiblePredictions} \]
\[ \land \quad \forall p' : \text{actual(flight) possiblePredictions} \]
\[ \quad \text{predicted(flight).timePeriod.size } \leq p'.\text{timePeriod.size} \]

The first condition states that for all flights, and predictions belonging to the flight's possible predictions and, at any time during the predictions time period, the following are true:

- The separation space of the prediction is always within the legal sector space.
- The separation space of the prediction at the end of the time period is within the last beacon.
- The altitude range of the separation space at the end of the time period is within the departure range of the flight.

The second condition states that for any planned flight:

- The predicted flight is a possible prediction of the flight.
- Every possible prediction of the flight takes at least as long as the predicted flight.
The next schema specifies the safety of the period:

\[
\begin{align*}
\text{ATCSafetyCondition} & \quad \quad \quad \quad \text{ATCWorkStructuresAttributes} \\
\quad \quad (\text{dom(collisions)} = \emptyset \land \text{unplannedExits} = \emptyset) & \quad \Rightarrow \text{safety} = \text{safe} \\
\quad \quad (\text{dom(collisions)} \neq \emptyset \lor \text{unplannedExits} \neq \emptyset) & \quad \Rightarrow \text{safety} = \text{unsafe}
\end{align*}
\]

The conditions state that the period is safe if there are no collisions and no unplanned exits and unsafe otherwise.
The next schema describes the estimated fuel consumption of all the planes in the sector:

\[
\text{ATCTotalFuelConsumptionCondition } \quad \text{ATCWorkStructuresAttributes}
\]

\[
\exists \text{ estimatedFuelConsumption: plannedFlights } \rightarrow \text{ Real } \quad \forall \text{ flight: plannedFlights } \quad \text{estimatedFuelConsumption(flight) = predicted(flight).fuelConsumption(predicted(flight).timePeriod.max)}
\]

\[
\wedge \text{ estimatedTotalFuelConsumption = sum(estimatedFuelConsumption) }
\]

The conditions state that there is a function, estimated fuel consumption, from the set of planned flights to the real numbers so that:

- The estimated fuel consumption for any planned flight is equal to the fuel consumption at the end of the predicted period of the flight.

- The total estimated fuel consumption is equal to the sum of estimated fuel consumption.
The final air traffic control period schema specifies the estimated total flying time for all the planes in the sector:

\[ \text{estimatedTotalFlyingTime} = \sum \text{estimatedFlyingTime} \]

\[ \forall \text{flight}: \text{plannedFlights} \bullet \\
\text{estimatedFlyingTime(flight)} = \text{actual(flight).timePeriod.size} + \text{predicted(flight).timePeriod.size} \]

The conditions state that there is a function, estimated flying time, from the set of planned flights to the real numbers so that:

- The estimated flying time for any planned flight is equal to the duration of the actual time period and the predicted time period.

- The total estimated fuel consumption is equal to the sum of estimated flying time.
The air traffic control period is described as the conjunction of the eleven schemas just described:

\[
\text{ATCWorkStructures} == \text{ATCFixedConditions} \land \text{ATCStartConditions} \land \text{ATCInsideSectorCondition} \land \text{ATCCollisionCondition} \land \text{ATCPlannedExitCondition} \land \text{ATCEnRouteCondition} \land \text{ATCUnplannedExitCondition} \land \text{ATCPredictedFlightCondition} \land \text{ATCSafetyCondition} \land \text{ATCTotalFuelConsumptionCondition} \land \text{ATCTotalFlyingTimeCondition}
\]

This section aimed to define a class of air traffic control work structures which consist of air traffic control scenarios, together with components which illustrate how the planned flights of the scenario are actualised in the control time period, together with predictions of what will happen to these flights after the control time period. An air traffic control scenario is an air traffic sector together with a set of planned flights, a control time period and a throughput. An air traffic sector is a three-dimensional space, together with a set of beacons and legal routes giving rise to corridors which connect beacons.
7.3 Desired Benefits

Air traffic control work domain is a sub-set of the transformations of air traffic work structures. In this section, a sub-set of the class of transformations of air traffic control work structures is defined and this sub-set is used to define the desired benefits.

The set of air traffic control transformations is defined in the following axiomatic description:

\[
\text{ATCDomain} \equiv \mathcal{P}([\text{ATCWorkStructures} \to \text{ATCWorkStructures}])
\]

\[
\forall \text{transformation} : \text{ATCDomain}; \text{structure} : \text{ATCWorkStructures} \cdot
\]

\[
\begin{align*}
&\text{structure.sector} = \text{transformation(structure).sector} \\
&\wedge \text{structure.beacons} = \text{transformation(structure).beacons} \\
&\wedge \text{structure.legalRoutes} = \text{transformation(structure).legalRoutes} \\
&\wedge \text{structure.corridorSpace} = \text{transformation(structure).corridorSpace} \\
&\wedge \text{structure.legalSectorSpace} = \text{transformation(structure).legalSectorSpace} \\
&\wedge \text{structure.minVertSep} = \text{transformation(structure).minVertSep} \\
&\wedge \text{structure.minTimeSep} = \text{transformation(structure).minTimeSep} \\
&\wedge \text{structure.speedLimit} = \text{transformation(structure).speedLimit} \\
&\wedge \text{structure.controlTimePeriod} = \text{transformation(structure).controlTimePeriod} \\
&\wedge \text{structure.plannedFlights} = \text{transformation(structure).plannedFlights} \\
&\wedge \text{structure.throughput} = \text{transformation(structure).throughput}
\end{align*}
\]

The air traffic control work domain is a sub-set of the set of all functions from air traffic control work structures to air traffic control work structures.
The condition states that each of the component attributes of the air traffic control scenarios remain unchanged in the transformation.

In order to define the air traffic control desired benefits, it is necessary to address the safe sector throughput, but this function was defined for sectors not for work structures. The partial function, safe throughput, extends the safe sector throughput function to definition on the set of all possible air traffic control work structures.

\[
\text{safeThroughput} : \text{ATCWorkStructures} \rightarrow \text{Real}
\]

\[
\forall \quad \text{sector} : \text{ATCSector} \quad \text{structure} : \text{ATCWorkStructures} \quad \cdot
\]

\[
\text{sector} = ( \quad \text{structure.sector}, \\
\text{structure.beacons}, \\
\text{structure.legalRoutes}, \\
\text{structure.corridorSpace} \\
\text{structure.legalSectorSpace}, \\
\text{structure.minVertSep}, \\
\text{structure.minTimeSep}, \\
\text{structure.speedLimit})
\]

\[
\Rightarrow \\
( \quad ( \quad \text{sector} \in \text{dom(safeSectorThroughput)} \\
\Leftrightarrow \\
\text{structure} \in \text{dom(safeThroughput)}) \\
\wedge \\
\text{safeThroughput(structure)} \\
= \text{safeSectorThroughput(sector)})
\]

The condition states that for all sectors and all work structures, if the sector attributes of the structure are equal to the attributes of the sector, then the following are true:

- The sector belongs to the domain of safe sector throughput if and, only if, the structure belongs to the domain of safe throughput.

- The safe throughput of the structure is equal to the safe sector throughput of the sector.
The air traffic control desired benefits are defined using an axiomatic description:

\[ \text{ATCDesiredBenefits} : \mathcal{P} \text{ATCDomain} \]

\[ \forall \text{transformation: ATCDesiredBenefits; structure : dom(safeThroughput);} \]
\[ \quad \text{structure.throughput} \leq \text{safeThroughput(structure)} \]
\[ \quad \Rightarrow \text{transformation(structure).safety} = \text{safe} \]

The air traffic control desired benefits are a sub-set of the air traffic control domain.

The condition states that for every desired benefit transformation and every work structure, if the structure's throughput is less than or equal to the safe throughput of that structure, then the desired benefit transformation must produce a safe air traffic control structure.

In this section, the air traffic control domain has been defined. This definition, in turn, has been used to define the air traffic control desired benefits.
7.4 The Benefit Ordering

In this last section, air traffic control benefit is defined using an axiomatic description:

\[
\text{ATCBenefit} : \text{PreOrdering}[\text{ATCDomain}]
\]

\[
\forall t,t' : \text{ATCDomain} \setminus \text{ATCDesiredBenefits} \cdot \\
\quad t =_{\text{ATCBenefit}} t'
\]

\[
\forall t : \text{ATCDomain} \setminus \text{ATCDesiredBenefits}; \\
\quad t' : \text{ATCDesiredBenefits} \cdot \\
\quad t <_{\text{ATCBenefit}} t'
\]

\[
\forall t,t' : \text{ATCDesiredBenefits} \cdot \\
\quad \forall s : \text{ATCWorkStructures} \cdot \\
\quad t \leq_{\text{ATCBenefit}} t' \quad \iff \quad \\
\quad \left( (t(s).\text{safety}, t(s).\text{estimatedTotalFuelConsumption}, t(s).\text{estimatedTotalFlyingTime}) \right) \leq_{(\text{ATCSafe}\times\text{Real}^0\times\text{Real}^0)} \\
\quad \left( (t'(s).\text{safety}, t'(s).\text{estimatedTotalFuelConsumption}, t'(s).\text{estimatedTotalFlyingTime}) \right)
\]
Air traffic control benefit is a pre-ordering of the ATC domain.

The conditions state that:

- Every pair of transformations which are not desired benefits are equivalent.

- Any desired benefit is a quality improvement on any transformation which is not a desired benefit.

- For transformations which have desired benefits, the benefit ordering is a cross-product of the pre-orderings of safety, estimated total fuel consumption and estimated total flying time.

The definition of ATC benefit presented in this section completes the definition of the components of performance requirements for air traffic control. The benefit requirements for air traffic control can be formally defined as follows:

\[
\text{ATCBenefitRequirements} == (\text{ATCDomain}, \text{ATCDesiredBenefits}, \text{ATCBenefit})
\]
7.5 Conclusion

This chapter has developed a specification of the benefit requirements for a formal language worksystem for Air Traffic Control. The specification was constructed to demonstrate that the performance requirements aspects of the formal framework were appropriate for likely Client Requirements. Although the benefit requirements specified here are undoubtedly a simplification of the likely requirements of air traffic authorities, they are sufficiently detailed to demonstrate that the performance requirements aspects of the formal framework are appropriate for Client Requirements. Thus, this chapter amounts to a further increment of knowledge at the level of the framework.

Chapters 6 and 7 have considered the specification of performance requirements for language worksystems. Chapters 8, 9 and 10 will consider the specification of language worksystems themselves. Chapter 8 considers the specification of language worksystems as a whole, whereas Chapters 9 and 10 concentrate on the specification of the syntax of languages.
8 Specifying Language Worksystems

This chapter considers the aspects of the formal framework concerned with specifying language worksystems. It is based upon the Conception of Formal Language Engineering developed in Chapter 4. The main thrust of the development of the formal framework of this thesis has been concerned with specifying desired performance for language worksystems and the syntax of languages. Due to these efforts, a characterisation of formal language worksystems that is on a par with the characterisation of desired performance, presented in Chapter 6, or language syntax presented in Chapter 9, has not been possible. Thus, this chapter defines a very simple case of a language worksystem. The definition is illustrated with an example paint shop worksystem that aims to satisfy the paint shop performance requirements established in Chapter 6. The approach to the general case is then considered. Chapters 9 and 10 consider the definition of language syntax in detail.

The first section formally defines the concept of a simple formal language. The definition is illustrated with two simple specifications of formal languages that could form the basis of a design for the paint shop worksystem whose requirements were specified in Chapter 6. The second section formally defines the concept of a simple language worksystem. The definition is illustrated by developing a corresponding specification of a simple language worksystem that satisfies the requirements for a paint shop worksystem outlined in Chapter 6. The final section considers the problems associated with designing language worksystems in general.
8.1 Simple Formal Languages

This section defines the notion of a simple formal language using the Z notation. The definition is illustrated with two specifications of formal languages for the paint shop worksystem.

Simple formal languages consist of syntax and denotational semantics. The syntax is a class of representations which is a sub-set of some syntactic structure class. Syntactic structure classes include things such as text, graphs, bitmaps, etc. The denotational semantics is the interpretation of the syntax in terms of the work structures of the domain. The denotational semantics of the simple formal language is such that each element of the syntax should give rise to a set of operations that could be performed upon a work structure.

A simple formal language defined over a class of work structures and a syntactic structure class is presented in the following Z schema. The schema consists of three components: the syntax a subset of the syntactic structure class; the null statement a distinguished item of syntax; and the denotational semantics, a partial function from syntax to relations defined on work structures.

\[
\text{SimpleFL} : [\text{WorkStructures}, \text{SyntacticStructureClass}] \rightarrow \text{syntax} : \mathcal{P}\text{SyntacticStructureClass} \\
\text{nullStatement} : \text{syntax} \\
\text{denotationalSemantics} : \text{syntax} \rightarrow (\text{WorkStructures} \leftrightarrow \text{WorkStructures})
\]

\[\text{dom(denotationalSemantics)} = \text{syntax} \setminus \{\text{nullStatement}\}\]

The condition states that the domain of the function denotational semantics is all of the syntax except the null element.

Two different formal languages are now specified. Both are potential languages for a formal language worksystem that will satisfy the paint shop performance requirements specified in Chapter 6. Both formal languages will
be based upon the syntactic structure class of text strings, introduced with the following global definition in Z:

[TextStrings]

8.2.1 Paint Shop Formal Language 1

The syntax of the first language will consist of just three strings. It is formally specified as follows:

\[
\text{FL1Syntax} = \{\text{"blue"}, \text{"yellow"}, \text{""}\}
\]

The null statement of the formal language will be the empty string. Formally:

\[
\text{FL1NullStatement} = \text{""}
\]

The denotational semantics of the formal language will map each syntactic element to a relation on work structures. The first element of the relation will be the work structure denoted by the language. The second element will be either the red work structure or the green work structure. Therefore, the first language describes the work structure and the possible transformations of the work structure that would achieve desired benefit according to the paint shop specification of benefit requirements.

Formally the denotational semantics function is specified using an axiomatic definition construct:

\[
\text{FL1DenotationalSemantics} : \text{FL1Syntax} \leftrightarrow (\text{PSWorkStructures} \leftrightarrow \text{PSWorkStructures})
\]

\[
\text{FL1DenotationalSemantics("blue") =}
((\text{blue,red}), (\text{blue,green}))
\]

\[
\text{FL1DenotationalSemantics("yellow") =}
((\text{yellow,red}), (\text{yellow,green}))
\]

The conditions state, by enumeration, the worksystem relations that are the denotational semantics of each work element.
The first formal language can now be specified by combining the syntax, null statement and denotational semantics.

\[
FL1 == (FL1Syntax, 
    FL1NullStatement, 
    FL1DenotationalSemantics)
\]

### 8.2.2 Paint Shop Formal Language 2

The syntax of the second language will consist of just three strings. It is formally specified as follows:

\[
FL2Syntax == \{"blue to red","yellow to red",""\}
\]

The null statement of the formal language will be the empty string. Formally:

\[
FL2NullStatement == ""
\]

The denotational semantics of the formal language will map each syntactic element to a singleton relation on work structures. In other words, the denotational semantics link each element of the syntax to a unique pair of work structures.

The denotational semantics is formally specified in terms of an axiomatic definition.

\[
\text{FL2DenotationalSemantics} : FL2Syntax \leftrightarrow \left(PSWorkStructures \leftrightarrow PSWorkStructures\right)
\]

\[
\text{FL2DenotationalSemantics}("blue to red") = \{(blue,red)\}
\]

\[
\text{FL2DenotationalSemantics}("yellow to red") = \{(yellow,red)\}
\]

The conditions state by enumeration the worksystem relations that are the denotational semantics of each work element. As can be seen from the
denotational semantics, the second language is more restrictive in that each non-null string implies a specific transformation of the work structures.

The second formal language can now be specified by combining the syntax, null statement and denotational semantics.

\[
FL2 == ( \text{FL2Syntax,} \\
\text{FL2NullStatement,} \\
\text{FL2DenotationalSemantics})
\]
8.2 Simple Language Worksystems

This section defines the notion of a simple language worksystem using the Z notation. The definition is illustrated with a specification of paint shop worksystems which relates to the specification of desired performance for the paint shop presented in Chapter 6.

Recall from Chapter 4:

'A formal language worksystem is composed of formal languages, as defined in Chapter 2, and language agents who interpret and employ the formal language to achieve the work benefits of the worksystem while incurring the worksystem costs.'
A sub-set of the class of all formal language worksystems is defined in the following Z schema:

---

\[
\text{SimpleFL Worksystem}[\text{WorkStructures, SyntacticStructureClass}]\
\]

\[
\text{SimpleFL}[\text{WorkStructures, SyntacticStructureClass}] \\
\text{recogniserAgent} : \text{PerformanceRequirements} \\
\quad [\text{WorkStructures} \times \text{syntax}] \\
\text{effectorAgent} : \text{PerformanceRequirements} \\
\quad [\text{WorkStructures} \times \text{syntax}] \\
\]

\[
\forall t \text{ recogniserAgent.domain} \Rightarrow \\
\quad \text{dom}(t) \subseteq \{w: \text{WorkStructures}; s: \text{syntax} \mid s = \text{nullStatement}\} \\
\]

\[
\forall t \text{ recogniserAgent.desiredBenefit} \Rightarrow \\
\quad \forall w, w' : \text{WorkStructures}, s, s' : \text{syntax} \Rightarrow \\
\quad \quad t(w, s) = (w', s') \\
\quad \quad \Rightarrow \\
\quad \quad w = w' \\
\]

\[
\forall t \text{ effectorAgent.domain} \Rightarrow \\
\quad \text{ran}(t) \subseteq \{w: \text{WorkStructures}; s: \text{syntax} \mid s = \text{nullStatement}\} \\
\]

\[
\forall t \text{ effectorAgent.desiredBenefit} \Rightarrow \\
\quad \forall w, w' : \text{WorkStructures}, s, s' : \text{syntax} \Rightarrow \\
\quad \quad t(w, s) = (w', s') \\
\quad \quad \Rightarrow \\
\quad \quad (w, w') \in \text{denotationalSemantics}(s) \\
\]
This worksystem consists of just one formal language and two language agents: a recogniser agent and an effector agent. The conditions state that:

- The recogniser agent transforms pairs which consist of work structures and null strings.
- The recogniser agent transformations have no effect upon work structures.
- The effector agent transformed pairs consist of a work structure and the null string.
- The transformation of work structures carried out by an effector agent must be consistent with the denotational semantics of the initial syntactic component.

To illustrate the notion of a formal language worksystem, we will attempt to specify a worksystem, PSWorksystem, that satisfies the paint shop requirements.

The PSWorksystem will have TextStrings as the syntactic structure class. Thus the following condition must hold

\[
\text{PSWorksystem} \in \text{SimpleFLWorksystem}[\text{PSWorkStructures,TextStrings}]
\]

Thus, PSWorksystem will consist of the components of a simple formal language worksystem. Formally, PSWorksystem can be specified as follows:

\[
\text{PSWorksystem} ==
(\text{PSSyntax,}
\text{PSNullStatement,}
\text{PSDenotationalSemantics,}
\text{PSRecogniserAgent}
\text{PSEffectorAgent})
\]
The formal language of the worksystem shall be FL1 specified in Section 8.1.1. Formally, this language can be specified by specifying the formal language components of the worksystem:

- \text{PSSyntax} == \text{FL1Syntax}
- \text{PSNullStatement} == \text{FL1NullStatement}
- \text{PSDenotationalSemantics} == \text{FL1DenotationalSemantics}

In order to complete the specification of PSWorksystem, it is necessary to define PSRecogniserAgent and PSEffectorAgent. In both cases, this need amounts to specifying performance requirements. The two specifications are contained in the following sections.

\textbf{8.2.1 PSRecogniserAgent}

PSRecogniserAgent will identify the colour of the bricks, and indicate this colour to the effectorAgent. It will perform this transformation at a cost of 2 units or less. From the definition of simple formal language worksystems, we know the following must hold:

\text{PSRecogniserAgent} \in \text{PerformanceRequirements[PSWorkStructures \times TextStrings]}

Thus PSRecogniserAgent will consist of the components of a performance requirement. Formally, PSRecogniserAgent is specified as follows:

\text{PSRecogniserAgent} ==
\hspace{1cm} (\text{PSRArea},
\text{PSRADesiredBenefits},
\text{PSRABenefit},
\text{PSRAPossibleCosts},
\text{PSRADesiredCosts},
\text{PSRACost},
\text{PSRADesiredPerformance},
\text{PSRAPerformance})
PSRADomain is specified as the class of transformations, which transform pairs consisting of a work structure and the null statement to pairs consisting of a work structure and any item of PSSyntax.

\[
\text{PSRADomain} : \mathcal{P} (\text{PSWorkStructures} \times \text{PSSyntax} \rightarrow \text{PSWorkStructures} \times \text{PSSyntax})
\]

\[
\forall t : \text{PSRADomain} .
\text{dom}(t) \subseteq \{w : \text{PSWorkStructures}; s : \text{PSSyntax} | s = \text{PSNullStatement}\}
\]

PSRADesiredBenefits is defined as a sub-set of PSRADomain containing the single transformation PSRAT. Where PSRAT is defined as follows:

\[
\text{PSRAT} = \{(\text{blue},""), (\text{blue},"blue"), (\text{yellow},""), (\text{yellow},"yellow")\}
\]

Thus, PSRAT represents the identification of the existing colour of the bricks. PSRADesiredBenefits can now be formally defined:

\[
\text{PSRADesiredBenefits} : \mathcal{P} \text{PSRADomain}
\]

\[
\text{PSRADesiredBenefits} = \{\text{PSRAT}\}
\]
Since the only transformation which has desired benefit is PSRAT, the specification of the benefit ordering consists of specifying that PSRAT is of greater benefit than all other transformations, and that all other transformations are equivalent. Formally:

\[
\text{PSRABenefit} : \text{PreOrdering[PSRADomain]}
\]

\[\forall t : \text{PSRADomain} - \text{PSRADesiredBenefits} \bullet
t <_{\text{PSRABenefit}} \text{PSRAT}\]

\[\forall t,t' : \text{PSRADomain} - \text{PSRADesiredBenefits} \bullet
t' =_{\text{PSRABenefit}} t\]

The conditions state that:

- PSRAT is of greater benefit than transformations which do not have desired benefits.
- All transformations which are not desired benefits are of equivalent benefit.

The cost structures for PSRecogniserAgent are the same as the cost structures for PSPerformanceRequirements. Formally:

\[
\text{PSRAPossibleCosts} == \text{PSPossibleCosts}
\]

PSRADesiredCosts are 2 units or less. Formally:

\[
\text{PSRADesiredCosts} == \{n \in \mathbb{N} \mid n \leq 2\}
\]

PSRACost ordering is the same as that for the PSPerformanceRequirements. Formally:

\[
\text{PSRACost} == \text{PSCost}
\]
PSRADesiredPerformance is the product of PSRADesiredBenefits and PSRADesiredCosts. Formally:

\[ \text{PSRADesiredPerformance} = \text{PSRADesiredBenefits} \times \text{PSRADesiredCosts} \]

PSRAPerformance is the product ordering of PSRABenefit and PSRACost. Formally:

\[ \text{PSRAPerformance} = \text{PSRABenefit} \times \text{PSRACosts} \]

**8.2.1 PSEffectorAgent**

PSEffectorAgent will transform a brick with its colour identified to either a red brick or a green brick. The cost of the transformation will be such that when it is combined with the transformation of the language recogniser it will achieve PSDesiredPerformance. From the definition of simple formal language worksystems, we know the following must hold:

\[ \text{PSEffectorAgent} \in \text{PerformanceRequirements[PSWorkStructures} \times \text{TextStrings}] \]

Thus, PSEffectorAgent will consist of the components of a performance requirement. Formally, PSEffectorAgent be specified as follows:

\[ \text{PSEffectorAgent} == \]

\[ (\text{PSEADomain}, \]
\[ \text{PSEADesiredBenefits,} \]
\[ \text{PSEABenefit,} \]
\[ \text{PSEAPossibleCosts,} \]
\[ \text{PSEADesiredCosts,} \]
\[ \text{PSEACost,} \]
\[ \text{PSEADesiredPerformance,} \]
\[ \text{PSEAPerformance}) \]
PSEADomain is specified as the class of transformations, which transform pairs consisting of a work structure and a syntax component identifying the work structure, to pairs consisting of a work structure and the PSNullStatement.

\[
PSEADomain : \mathcal{P} \left( \text{PSWorkStructures} \times \text{PSSyntax} \rightarrow \text{PSWorkStructures} \times \text{PSSyntax} \right)
\]

\[
\forall t : \text{PSEADomain} \bullet \\
\text{rant}(t) \subseteq \{ w : \text{PSWorkStructures}; s : \text{PSSyntax} \mid s = \text{PSNullStatement} \}
\]
PSEADesiredBenefits is defined as a sub-set of PSEADomain. In order to make this definition and the PSEABenefit ordering simpler, an auxiliary function, ProsDesBen, is introduced which is a bijection that projects the transformations of PSEADesiredBenefit to PSDesiredBenefit.

\[
\begin{align*}
PSEADesiredBenefits & : \mathcal{P} PSRADomain \\
ProjDesBen & : PSEADesiredBenefits \rightarrow PSDesiredBenefits
\end{align*}
\]

\[
\forall t : PSEADesiredBenefits \bullet \\
\quad \text{dom}(t) = \text{ran}(PSRAT)
\]

\[
\forall t : PSEADesiredBenefits \bullet \\
\quad \forall w : PSWorkStructures, s, s' : PSSyntax \bullet \\
\quad \quad t(w, s) = (\text{ProjDesBen}(t)(w), s')
\]

\[
\forall t : PSEADesiredBenefits \bullet \\
\quad \forall w, w' : PSWorkStructures, s, s' : PSSyntax \bullet \\
\quad \quad t(w, s) = (w', s') \\
\quad \quad \Rightarrow \\
\quad \quad (w, w') \in PSDenotationalSemantics(s)
\]

The conditions state that:

- The domain of any desired benefit transformation of the effector agent is the range of PSRAT, the only desired benefit of the recogniser agent.

- For any desired benefit transformation of the effector agent, there is a corresponding desired benefit transformation, ProjDesBen, from the paint shop desired benefits, so that the effector agent transformation performs the paint shop transformation on the paint shop worksystems work structure component.

- The transformation of work structures carried out by the paint shop effector agent must be consistent with the paint shop denotational semantics of the initial syntactic component.
The PSEABenefit ordering is derived from the paint shop benefit ordering. Formally:

\[
PSEABenefit : \text{PreOrdering}[\text{PSEADomain}]
\]

\[
\forall t, t' : \text{PSRADomain} - \text{PSRADesiredBenefits} \quad t \leq_{PSEABenefit} t'
\]

\[
\forall t, t' : \text{PSRADomain} - \text{PSRADesiredBenefits} \quad t =_{\text{PSRBenefit}} t'
\]

\[
\forall t, t' : \text{PSEADesiredBenefits} \quad t \leq_{\text{PSEABenefit}} t' \implies \text{ProjDesBen}(t) \leq_{\text{ProjDesBen}} \text{ProjDesBen}(t')
\]

The conditions state that:

- Any transformation which is a desired benefit improves upon any transformation which is not a desired benefit.
- All transformations which are not desired benefits are of equivalent benefit.
- For all transformations which are desired benefit for the effector agent, the benefit ordering is derived from the paint shop benefit ordering through the auxiliary function ProjDesBen.

The cost structures for PSEffectorAgent are the same as the cost structures for PSPerformanceRequirements. Formally:

\[
PSEAPossibleCosts =\text{PSPossibleCosts}
\]
Since the recogniser has incurred 2 cost units and the desired costs of the paint shop as a whole are 4 or less, \( \text{PSEADesiredCosts} \) are 2 units or less. Formally:

\[
\text{PSEADesiredCosts} = \{ n \in \mathbb{N} \mid n \leq 2 \}
\]

\( \text{PSEACost} \) ordering is the same as that for the \( \text{PSPerformanceRequirements} \). Formally:

\[
\text{PSEACost} = \text{PSCost}
\]

Recall from the paint shop performance requirements specification:

\[
\text{PSDesiredPerformance} = \{ t : \text{PSDesiredBenefits}; c : \text{PSDesiredCosts} \mid \\
\neg (\text{ran}(t) = \{\text{green}\} \land c = 4) \}
\]

Since two of the cost units have been accounted for by the recogniser agent, the following is the formal definition of effector agent desired performance:

\[
\text{PSEADesiredPerformance} = \{ t : \text{PSEADesiredBenefits}; c : \text{PSEADesiredCosts} \mid \\
\neg (\text{ran}(t) = \{(\text{green},\"\")\} \land c = 2) \}
\]

\( \text{PSEAPerformance} \) is a super-set of the product ordering of \( \text{PSRABenefit} \) and \( \text{PSRACost} \). Formally:

\[
\text{PSEAPerformance} : \text{PreOrdering} [\text{PSEADomain} \times \text{PSEAPossibleCosts}]
\]

\[
\text{PSEABenefits} \times \text{PSEACosts} \subseteq \text{PSEAPerformance}
\]

\[
\forall t,t' : \text{PSEADomain}; c,c' : \text{PSEAPossibleCosts} \\
((t,c) \not\in \text{PSEADesiredPerformance} \\
\land \\
(t',c') \not\in \text{PSEADesiredPerformance}) \\
\Rightarrow \\
(t,c) \Rightarrow \text{PSEAPerformance} (t',c')
\]
The conditions state that:

- The product of the paint shop benefit and cost orderings is a sub-set of the paint shop performance ordering.

- Any combination of work transformations and cost structures that do not exhibit desired performance are equal in the performance ordering.
8.3 Language Worksystems in General

This section considers the generalisation of the notion of language worksystem from the simple case considered in the sections earlier. It contains the following sub-sections:

- Language Syntax
- Language Semantics
- Allowable Agent Transformations
- Agent Combination

The first section considers language syntax and how a general notion of a worksystem should deal with language syntax. The second section considers language semantics. It considers different existing styles of semantic definitions and outlines how these relate to formal language worksystems. It considers the role of denotational semantics as part of language worksystems. The third section considers a possible deficiency in the notion of a work domain as it currently stands. The final section considers how to deal with the issue of agent combination or, in other words, how to determine whether a specified language worksystem achieves desired performance.

8.3.1 Language Syntax

In the simple language worksystem, the syntax of the language was a sub-set of some given syntactic structure class. In general, the design of language worksystems will involve a choice of syntactic structure class and the determination of syntax within that syntactic structure class.

The choice of syntactic structure class will be chosen from a generalised class of syntactic structure classes. Ideally, it would be possible to combine a number of syntactic structure classes to create a new class. Any general notion of formal language worksystem will therefore require a characterisation of such a class of syntactic structure classes. Some efforts in this direction are reported in Chapters 9 and 10.
Further, mechanisms will be required for defining the syntax of a language as a sub-set of the chosen syntactic structure class. In the text string class, grammatical formalisms based upon production rules are used for this purpose. Chapter 9 presents a generalisation of the notion of grammatical formalism for this purpose.

### 8.3.2 Language Semantics

In traditional approaches to programming language semantics, there are three forms of semantic description:

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>denotational</td>
<td>Denotational semantics attempt to define the semantics of programming languages by mapping the syntax to special mathematical constructions known as domains [Stoy 77]. The term was adopted in the simple language worksystem to define the relationship between the syntax and the work domain.</td>
</tr>
<tr>
<td>operational</td>
<td>Operational semantics attempt to define the meaning of programming languages in terms of some computing machine behaviour, either some physical machine or an abstract simplification. In the simple language worksystem, operational semantics may be thought of as the benefit requirements for the language agents.</td>
</tr>
<tr>
<td>axiomatic</td>
<td>Axiomatic semantics defines the semantics of languages in terms of axioms and rules of inference. By applying the rules of inference to the axioms, it is possible to determine the equivalence of syntactic elements. Axiomatic semantics might be thought of as arising in formal language worksystems when there are two or more languages in the worksystem. Any relation between the semantics of languages forms an axiomatic semantics for the two languages.</td>
</tr>
</tbody>
</table>

In the specification of programming languages, a denotational semantic description is seen as being more abstract than an operational semantics.
description. In other words, the relationship between denotational semantics and operational semantics is of the class instance form. Thus, denotational semantics is a specification for operational semantics. Establishing such a class instance relationship between denotational and operational semantics was the intention behind the definition of the denotational semantic function of the simple language worksystem of the previous section. It was intended that the definition of the formal languages would act as an intermediate stage between the performance requirements and the specification of the whole worksystem. In the simple formal language worksystem specified in the previous section, the denotational semantics function mapped the syntax to a relation on work structures. If the notion of denotational semantics is carried forward into a general definition of language worksystems, the form of a generalised denotational semantic relation would have to be determined.

The view of denotational semantics outlined above is not the only possible interpretation. An alternative view could equate the denotational semantics of a language with benefit requirements of an agent and the operational semantics with the artifact specification of that agent. This view does not attempt to generalise the notion of denotational semantics to language agents, but rather views denotational semantics as the benefit requirements for an interpreter specifying the transformations the interpreter is required to carry out. This view is attractive, since it does not add a (possibly unnecessary) extra level of abstraction.

8.3.3 Basic Transformations

This sub-section considers the decomposition of requirements specifications into component requirement specifications and at what stage the decomposition can stop.

Given some statement of performance requirements (as defined in Chapter 6), it is possible to distinguish between the different transformations in the domain. Some transformations might be considered basic, in the sense that they can be carried out directly by the worksystem.

The concept of basic transformations can be exemplified by considering the work domain for air traffic control. Thus, basic transformations in the air
traffic control domain include those which just alter the speed or flight level of the individual aircraft. Since these transformations can be carried out directly by an air traffic control worksystem. It is only through basic transformations that the air traffic control worksystem can alter the other attributes of ATCWorkStructures, which relate to safety, flying time and fuel consumption.

The decomposition of Performance Requirements for a worksystem as a whole into a collection of formal languages and the Performance Requirements for a set of agents, may be thought of as being completed when all of the agents perform only basic transformations of work structures.

Worksystems cannot alter directly the safety or fuel consumption of the aircraft. Transformations that can be carried out directly might be called basic transformations. In one sense worksystem decomposition is completed when every agent is specified in terms of basic transformations of the work domain. To capture this notion, the definition of a work domain given in Chapter 6 would have to be extended.

8.3.4 Agent Combination

The example simple language worksystem fails to describe formally how the agents are combined to achieve desired performance. This is for a simple case with two agents: a recogniser and an effector. For the general case, we require a means for combining multiple agents. An approach that might be useful is based upon category theory.

Consider a diagram in some category which has an arrow going from an object called work structures to itself and the arrow denotes the performance requirements.

\[
\text{Workstructures} \xrightarrow{\text{Performance requirements}} \text{Workstructures}
\]
A worksystem similar to the simple formal language worksystem might be represented in the same fashion:

Here, the recogniser performance requirements are represented as an arrow from the work structures object to an object representing the language syntax and work structures combined. The effector performance requirements are represented as an arrow from the language syntax and work structures object to the work structures object.

In a simple sense, the worksystem might be said to achieve desired performance if the following diagram commutes:
The problem is how to generalise the simple worksystem to the general case. It might be possible to address this problem through the concept of co-cones. Consider the following diagram, D:

Any co-cone for D must consist of an object d, together with injection arrows da, db, dc such that, the following diagram commutes:
Consider the following diagram:

If this diagram commutes, then the following assignment is a co-cone of D.
\[ \begin{align*}
  d &= c \\
  da &= h \\
  db &= g \\
  dc &= \text{id}(c)
\end{align*} \]

Since the following diagram commutes:

Consider the case: where a and c represent the class of work structures; b a combination of work structures and language elements; and f and g the desired performance of language agents. Then, the worksystem would achieve desired performance h, if there was a co-cone whose object was c and with h the injection from a to c.

It might be possible to generalise the concept of language worksystems, by generalising the worksystem diagram so that a worksystem represented by a diagram D with two objects representing the classes of work structures ws and ws'. Then, a worksystem might be said to achieve desired performance, if there was a co-cone whose object was ws', and the injection from ws to ws' was the performance requirements of the worksystem.
This conjecture is somewhat fanciful. To validate the conjecture it would be necessary to construct a category whose objects were combinations of classes of work structures and language syntax and whose arrows were performance requirements. This construction would necessitate a major redefinition of the performance requirements.

Another way of combining language agents might be to extend the process algebras such as CCS\[Milner 89\] and CSP\[Hoare 85\], which are concerned with the composition of agents. Even if the categorial approach is attempted first, these formalisms might shed a great deal of light on the problem of combining agents.
8.4 Conclusion

The conclusion to this chapter summarises the chapter and its achievements and assesses these achievements by the criteria established in the introduction (Chapter 1).

8.4.1 Summary

This chapter has considered the aspects of the formal framework concerned with specifying language worksystems. Using Z, the concept of a simple formal language was specified and simple formal languages, which relate to the paint shop example of Chapter 6, were developed in line with the concept. The concept of a simple formal language worksystem was specified as consisting of a simple formal language, a recogniser agent and an effector agent. Where both the agents were described as Performance Requirements whose work structures consist of the work structures of the domain and the syntactic elements of the formal language. A simple example of a paint shop worksystem was developed in line with the simple formal language worksystem concept. It was demonstrated that the example worksystem satisfied the paint shop performance requirements established in Chapter 6.

In the final section, the approach to the general case was considered. This section raised the issues of language syntax, language semantics, basic transformations and agent combination. The issue of language syntax is discussed further in Chapters 9 and 10. Different perspectives on semantics might lead to different forms of a generalised specification formal language worksystems. The issues of basic transformations and agent composition both indicate that the concept of performance requirements specified in Chapter 6 may need to be extended as part of the attempt to develop a general characterisation of formal language worksystems.
8.4.2 Assessment

Recall that Chapter 1 outlined the following criteria for assessing the thesis that are applicable to the framework aspects of the thesis:

- Internal consistency between conception and framework
- External synthesis at the level of the framework
- Knowledge incremented at the level of the framework

In this chapter, the conceptual definition of a worksystem from Chapter 4 formed the basis for the formal definition of the simple formal language worksystem. Thus, it can be claimed that the worksystem aspects of the framework, as far as they have been specified, are consistent with the conception. The worksystem aspects of the framework, however, are incomplete. Further, the work of Section 8.3 indicates that the completion of the worksystem aspects of the framework might require redefinition of the performance requirements aspects of the framework in order to maintain the internal consistency of the framework.

As far as is known, the concept of performance requirements developed in Chapter 6, based upon a pre-ordering of work structure transformations, is a novel approach to requirements. Hence, the concept of a language worksystem whose agents are defined in terms of performance requirements must also be novel and, thus, an increment to existing knowledge at the level of the framework. The work concerned with extending the specification of the worksystem concept to the general case, although largely at the level of conjecture, can be seen as exploring potential avenues by which knowledge may be incremented at the level of the framework.

The ideas put forward in this chapter are clearly a result of a synthesis of a great deal of work from fields such as software engineering and cognitive ergonomics. However, as was pointed out in the introduction to this chapter, a characterisation of formal language worksystems that is on a par with, for example, the characterisation of desired performance presented in Chapter 6, has not been possible. Consequently, the thesis does not explicitly draw out the degree of synthesis, at the level of the worksystem aspects of the framework, as much as might have otherwise been the case. As has already
been mentioned, Chapter 9 considers a particular aspect of language worksystems, that of syntax, at a much greater level of detail. With the level of detail of this chapter, it is possible to make the degree of synthesis much more explicit.
9 Specifying Language Syntax

This chapter develops the aspects of the formal framework concerned with defining the syntax of languages. It is an instantiation of the Conception of Formal Language Engineering developed in Chapter 4. It employs the basic concepts of Z and category theory.

The first section introduces a number of existing grammatical formalisms. These include text-based grammars, shape grammars, graph grammars and picture layout grammars. Each of the grammars of the first section are concerned with different syntactic structures. The second section attempts to formalise this notion of syntactic structure for each of the grammars of the first section.

The third section develops a general notion of syntactic structure class. The completeness and consistency of this generalisation is discussed, using the different classes of syntactic structure outlined in Section 2. The fourth section employs the generalised notion of syntactic structure class to develop the notions of terminal and non-terminal categories. The concepts of terminal and non-terminal categories are then used in the fifth section to construct a generalised notion of grammar and derivation. The consistency and completeness of the generalisations of grammar and derivation are considered by outlining derivations for each of the grammars considered in Sections 1 and 2.

The sixth section employs the generalised notions of syntax and grammar developed in Sections 3 to 5 to develop a novel grammatical formalism for specifying sub-sets of Venn grammars.
9.1 Existing Grammatical Formalisms

This section introduces a number of existing grammatical formalisms. It contains the following sub-sections:

- Text-Based Grammars
- Shape Grammars
- Graph Grammars
- Picture Layout Grammars
- Other Work

Each of the sections introduces grammatical formalisms for defining the syntax of languages in a different syntactic medium.

9.1.1 Text-Based Grammars

This section introduces a production rule formalism for defining the syntax of text-based languages. Further details of production rule formalisms can be found in Hopcroft and Ullman [Hopcroft and Ullman 79]

Grammatical formalisms consist of, in part, languages for defining the syntax of another language. The language component of a grammatical formalism can be conveniently called the meta language. It is important to distinguish the constructs of the language from those of the meta language.

The constructs of the meta language are of two sorts. There are the meta symbols, which are used to express the production rule, and there are the syntactic structures. The syntactic structures have terminal and non-terminal components. For text-based production rules the only meta symbol is the arrow:

-->
The syntactic structures consist of strings of characters, where each character is either terminal or non-terminal. For the purpose in hand the terminal characters might be thought of as the digits:

0, 1, ..., 9

The non-terminal characters may be thought of as the upper-case letters of the Roman alphabet:

A, ..., Z

The following is an example of a grammar. It defines a class of binary numerals with text-based production rules:

<table>
<thead>
<tr>
<th>Rule Number</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B → D</td>
</tr>
<tr>
<td>2</td>
<td>B → DB</td>
</tr>
<tr>
<td>3</td>
<td>D → 0</td>
</tr>
<tr>
<td>4</td>
<td>D → 1</td>
</tr>
</tbody>
</table>

The numbers to the right are not part of the grammar but are to assist in the derivations presented below. The non-terminal symbol B is used as a start symbol which denotes an arbitrary binary numeral.

The following strings can be generated by the grammar:

0 10 010

Their generation can be illustrated by providing derivations of the strings. These derivations consist of sequences of non-terminal strings that begin with the start symbol and end with the required numeral. Each item in the sequence is the result of applying a production rule to a non-terminal character of the previous string. Derivations are now presented for each of the example strings above.
The derivation for 0 is as follows:

```
B
D   by production rule 1
0   by production rule 3
```

The derivation for 10 is as follows:

```
B
DB  by production rule 2
DD  by production rule 1
1D  by production rule 4
10  by production rule 3
```

The derivation for 010 is as follows:

```
B
DB  by production rule 2
DDB by production rule 2
DDD by production rule 1
ODD by production rule 3
OID by production rule 4
010 by production rule 3
```

### 9.1.2 Shape Grammars

Shape grammars were developed by [Stiny 80] as a means for defining languages of two- and three-dimensional spatial designs. The meta language was developed in the field of Architectural Design Theory and has been used to express classes of architectural design varying from the Prairie houses of Frank Lloyd Wright [Koning & Eizenberg 81] to Mughul gardens [Stiny & Mitchell 80].

The elements of shape grammars are straight lines. Shapes are formed with a spatial combination of lines. A shape is defined as a finite set of lines, no two of which can combine to form another line.
The following is an example of a shape grammar:

The grammar consists of two rules. The arrow is a meta character of the language. The triangular shape marked with dot • is used to mark the non-terminal shape. The plain triangular shapes are the terminal symbols. A shape matches with another shape if it is a (linear) transformation of that shape. The following three shapes are all generated by the shape grammar:

The first shape is generated by the second rule. The second shape is generated by the first rule followed by the second rule. The third shape is generated by the first rule twice followed by the second rule.

The shape grammar formalism outlined above has been extended [Stiny 80], to allow the definition of shapes parameterised in terms of co-ordinate variables. Parameterised shape grammars allow the definition of more generalised classes of shapes, such as the class of all quadrilaterals.
9.1.3 Graph Grammars

The notion of a graph grammar has been applied in fields as diverse as software engineering and biology (see [Panel 91]). Graph grammars attempt to define classes of graphs, where a graph is a set of objects together with a set of arrows which connect the objects. There are a variety of different approaches to graph grammars:

- Programmed Attribute Graph Grammars (PAGGs) [Gottler 92]
- Hyperedge Replacement [Drews & Kreowski 91]
- Node Label Controlled (NLC) Graph Grammars [Engelfreit & Rozenberg 91]
- Algebraic Approach to Graph Grammars [Ehrig et al. 91]

The algebraic approach is distinct from the other three, in that it deals with the graph grammars at a higher level of abstraction. Indeed, the first approach has been embedded in the algebraic approach [Barthelmann 91]. The algebraic approach, with its use of categories, has been a strong influence on the generalised notion of grammar developed in this chapter.

The Hyperedge Replacement approach is used to give an example of a graph grammar. The example is taken from the literature [Pavlidis 72], and generates Two Terminal Series Parallel Network (TTSPN). The objects are represented with black dots. The Hyperarrow (rather than hyperedge) is the non-terminal and is represented with numbered labelled arcs going to and from a square label.
The grammar has three production rules.

```
The following are example TTSPNs:
```

![Diagram of TTSPNs]

259
The following, although a network, is not a TTSPN:

![Diagram of a network](image-url)

The rather odd choice of a class of graphs to generate (i.e. TTSPNs rather than all Networks) indicates that many common classes of graphs cannot be generated by a context-free grammar such as the one above.

### 9.1.4 Picture Layout Grammars

Picture Layout Grammars, [Golin & Reiss 89], were designed for the definition of the syntax of Visual Programming Languages.

The graphical elements of a picture layout grammar are (computationally) primitive graphical objects, such as lines, shapes, and text strings. The primitive graphical objects are combined to form pictures. A collection of pictures forms the syntax of a visual language. Picture layout grammars, therefore, attempt to define a class of pictures.

The meta language for picture layout grammars is text. Thus, certain text strings are used to represent non-terminals. Functional notation is used to represent the combination of picture elements. A set of strings distinct from the non-terminals is used to represent basic shapes. The basic shapes are the terminals of the formalism. The following examples illustrate the use of the functional notation:

- `over(B,C)`  
  B is over C
- `left_of(B,C)`  
  B is to the left of C
- `contains(B,C)`  
  B contains C
- `follow(B,C)`  
  the right end of B is the left end of C
A class of towers can be defined, using a picture layout grammar and a basic element called a Block, which is represented below:

\[
\begin{array}{c}
\end{array}
\]

The grammar for a tower consists of the following two production rules:

\[
\begin{align*}
\text{Tower} & \rightarrow \text{Block} \\
\text{Tower} & \rightarrow \text{over}(\text{Block}, \text{Tower})
\end{align*}
\]

The following pair of objects are both towers:

\[
\begin{array}{cc}
\begin{array}{c}
\end{array} & \begin{array}{c}
\end{array} \\
\begin{array}{c}
\end{array} & \begin{array}{c}
\end{array}
\end{array}
\]

### 9.1.5 Other Work

The examples above are drawn from three research areas:

- Architectural Design Theory
- Graph Grammars
- Visual Programming

The work of Stiny on shape grammars is the least-developed of these areas. Indeed, literature reviews have uncovered no recent work on shape grammars. The other two areas appear to be thriving, both holding regular workshops.

Other work concerned with the specification of non-text based languages includes work on Array and Tree Grammars [Gips & Stiny 80].
9.2 Syntactic Structures

In Sub-section 9.1.1 the notion of meta language is introduced in the discussion of text-based production rules. A meta language is said to consist of two sorts of construct: the meta symbols which are used to describe the production rules, and the syntactic structures. The syntactic structures are the constructs that make up the left- and right-hand sides of the production rules.

The sub-sections that follow, use the Z notation to describe the nature of the syntactic structures for each of the grammatical formalisms considered above. Each formal description consists of three schemas which describe:

Components
Terminal Relationships
Non-Terminal Relationships

The first schema describes the basic components of the syntactic structure, to which production rules are applied. The second schema describes the terminal relationships that are preserved by production rules. The third schema describes the non-terminal relationships that are preserved when performing the matching that is part of production rule application.

This section contains the following sub-sections:

Text-based Grammars
Shape Grammars
Graph Grammars
Picture Layout Grammars
9.2.1 Text-based Grammars

The syntactic structures of text-based grammars are strings of terminal and non-terminal characters. The basic components of any string are the different positions within a string. Formally, this can be defined with a Z schema, generic over a class of positions, CPositions. The schema consists of a single attribute positions, which is a finite sub-set of CPositions:

\[
\text{TStringComponents}[\text{CPositions}]
\]

\[
\begin{align*}
\text{positions} & : F \text{ CPositions} \\
\end{align*}
\]

The schema has no conditions.

The terminal relationships of a string are concerned with the ordering of the positions in the string, and the position of various terminal characters in the string. The terminal relationships are formally defined in the schema T$string below. This schema is generic over the classes CPositions and CAlphabet. It consists of the schema TStringComponents over CPositions, together with two additional components. The after component is an infix relation between positions. The character component is a partial function, mapping positions to elements of the class CAlphabet.

\[
\text{TString}[\text{CPositions,CAlphabet}]
\]

\[
\begin{align*}
\text{TStringComponents}[\text{CPositions}] \\
\_after_ & : \text{positions} \leftrightarrow \text{positions} \\
\text{character} & : \text{positions} \rightarrow \text{CAlphabet} \\
\end{align*}
\]

\[
\forall \ p1,p2,p3 : \text{positions} \cdot \\
\neg \ p1 \, \text{after} \, p1 \\
\land \\
(p1 \neq p2 \Rightarrow p2 \, \text{after} \, p1 \lor p1 \, \text{after} \, p2) \\
\land \\
(p3 \, \text{after} \, p2 \land p2 \, \text{after} \, p1 \Rightarrow p3 \, \text{after} \, p1)
\]

263
The condition states that:

- the after relationship is irreflexive
- for any two non-equal positions, one must be after the other.
- The after relationship is transitive.

The non-terminal relationships of a string are concerned with a stricter ordering of the positions in the string and the position of various non-terminal characters in the string. The non-terminal relationships are formally defined in the schema NTString below. This schema is generic over the classes CPositions, CAlphabet and CMAlphabet. It consists of the schema TString over CPositions and CAlphabet, together with two additional components. The next component is a partial function from positions to positions. The nTCharacter component is a partial function mapping positions to elements of the class CMAlphabet.

\[
\text{NTString} \quad \text{[CPositions,CAlphabet,CNAlphabet]}
\]

\[
\text{TString[CPosition,CAlphabet]}
\]

\[
\begin{align*}
\text{next} & : \text{positions} \leftrightarrow \text{positions} \\
\text{nTCharacter} & : \text{positions} \rightarrow \text{CMAlphabet}
\end{align*}
\]

\[
\text{dom(next)} = \text{ran(after)}
\]

\[
\forall p1,p2 : \text{positions} \quad
\text{next}(p1) \text{ after } p1 \\
\wedge \\
(p1 \neq p2 \Rightarrow p1 \text{ after } p2 \lor p2 \text{ after } \text{next}(p1))
\]

\[
\text{dom(character)} \cup \text{dom(nTCharacter)} = \text{positions}
\]

\[
\text{dom(character)} \cap \text{dom(nTCharacter)} = \emptyset
\]
The conditions state that:

- The domain of the function is the range of the relation after. In other words, the domain of the function next is all of the positions, except the last one.
- The next position is after a position, and there is no position that lies between the position and its next position.
- The domain of the character and nTCharacter functions consists of all the positions.
- There is no intersection between the domain of the character and nTCharacter functions.

In defining a class of actual NTStrings it is necessary to define the actual generic parameters of the schema. These are presented in the following table:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Class Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positions</td>
<td>{p_1, p_2, ..., p_n}</td>
</tr>
<tr>
<td>Alphabet</td>
<td>{0, 1, ..., 9}</td>
</tr>
<tr>
<td>NAlphabet</td>
<td>{A, ..., Z}</td>
</tr>
</tbody>
</table>

The string ODB can be represented as a member of the class:

\[ \text{NTString}[^\{\text{Positions, Alphabet, NAlphabet}\}] \]

This membership is illustrated by constructing a table which lists attribute names and the values for the string ODB:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>positions</td>
<td>{p_1, p_2, p_3}</td>
</tr>
<tr>
<td><em>after</em></td>
<td>{(p_2, p_1), (p_3, p_1), (p_3, p_2)}</td>
</tr>
<tr>
<td>character</td>
<td>{(p_1, 0)}</td>
</tr>
<tr>
<td>next</td>
<td>{(p_1, p_2), (p_2, p_3)}</td>
</tr>
<tr>
<td>nTCharacter</td>
<td>{(p_2, D), (p_3, B)}</td>
</tr>
</tbody>
</table>

This description is less than concise. However, the detail is necessary in order to achieve a generalised notion.
9.2.2 Shape Grammars

The syntactic structures of shape grammars have as basic components a collection of lines. Formally, the class of these components can be defined with a Z schema, TShapeComponents, generic over a class of lines CLine. The schema consists of a single attribute lines, which is a finite sub-set of CLine:

\[
\text{TShapeComponents} : [\text{CLine}] \\
\text{lines} : \mathbb{F} \text{CLine}
\]

The schema has no conditions.

The terminal relationships of a string are concerned with the position of the string in two-dimensional Euclidean space. The terminal relationships are formally defined in the schema TShape below. This schema is generic over the class CLine. It consists of the schema TShapeComponents over CLine, together with five additional components. The hStart, hEnd, vStart, vEnd components are all total functions from the component lines to the Real numbers (formally defined in Chapter 8). The components represent, respectively, the horizontal start and end, and the vertical start and end of the line. The values component is a function mapping lines to a set of points in two-dimensional Euclidean space.
The conditions state:

- For any line, the horizontal start is always less than or equal to the horizontal end.
- The set of all points on the line forms the values of the line.
- Any two lines intersect at not more than one point.

The first condition ensures that lines are uniquely described. The final condition ensures that any shape is described by a unique set of lines.

The non-terminal relationships of a shape are concerned with tagging certain sets of lines with non-terminal symbols. The non-terminal relationships are formally defined in the schema NTShape below. This schema is generic over the classes CLine and CTag. It consists of the schema TShape over CLine,
together with an additional component tag, which is a partial function from the sub-sets of the component lines to the class CTag.

\[
\text{NTShape} \subseteq [\text{CLine}, \text{CTag}]
\]

\[
\text{TShape}[\text{CLine}]
\]

\[
tag : [\mathcal{P}\text{ lines} \rightarrow \text{CTag}]
\]

\[
\forall L1, L2 : \text{dom}(tag) \cdot L1 \cap L2 = \emptyset
\]

The condition states that any two sets of tagged lines do not intersect.

As for strings, in defining a class of actual NTShapes, it is necessary to define the actual generic parameters of the schema. These parameters are presented in the following table:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Class Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>{l1, l2, \ldots}</td>
</tr>
<tr>
<td>Tag</td>
<td>{\cdot, *, @, \ldots}</td>
</tr>
</tbody>
</table>

Any shape can now be represented as an element of the class

\[
\text{NTShape}[\text{Line, Tag}]
\]

Consider, for example, the following shape, drawn with horizontal and vertical axes and lines labelled l1 to l6:

![Diagram of a shape with labeled lines](image-url)
It can be demonstrated how this shape can be viewed as a member of the class by constructing a table which lists attribute names and the values for the shape:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>{11,12,13,14,15,16}</td>
</tr>
<tr>
<td>hStart</td>
<td>{(11,0), (12,0), (13,2), (14,1), (15,2), (16,1)}</td>
</tr>
<tr>
<td>hEnd</td>
<td>{(11,2), (12,4), (13,4), (14,2), (15,3), (16,3)}</td>
</tr>
<tr>
<td>vStart</td>
<td>{(11,2\sqrt{3}), (12,2\sqrt{3}), (13,0), (14,\sqrt{3}), (15,2\sqrt{3}), (16,\sqrt{3})}</td>
</tr>
<tr>
<td>vEnd</td>
<td>{(11,0), (12,2\sqrt{3}), (13,2\sqrt{3}), (14,2\sqrt{3}), (15,\sqrt{3}), (16,\sqrt{3})}</td>
</tr>
<tr>
<td>values</td>
<td>details omitted</td>
</tr>
<tr>
<td>tag</td>
<td>{([14,15,16], , )}</td>
</tr>
</tbody>
</table>
9.2.3 Graph Grammars

The syntactic structures of graph grammars have, as basic components, a collection of 'components' which will represent objects, arrows and hyperarrows. Formally, these components can be defined with a Z schema, TShapeComponents, generic over a class of lines CComponents. The schema consists of a single attribute, components which is a finite sub-set of CComponents:

\[
\text{TGraphComponents } [\text{CComponents}]
\]

\[
\text{components : } \mathcal{P} \text{CComponents}
\]

The schema has no conditions.

The terminal relationships of a graph are concerned with determining the objects and the arrows, and the source and target of those arrows. The terminal relationships are formally defined in the schema TGraph below. This schema is generic over the class CComponents. It consists of the schema TGraphComponents over CComponents, together with four additional components. The objects and arrows components are sub-sets of the 'components' component. The source and target are total functions from the arrows component to the objects component.

\[
\text{TGraph } [\text{CComponents}]
\]

\[
\begin{align*}
\text{TGraphComponents}[\text{CComponents]} \\
\text{objects} : \mathcal{P} \text{components} \\
\text{arrows} : \mathcal{P} \text{components} \\
\text{source} : \text{arrows } \rightarrow \text{objects} \\
\text{target} : \text{arrows } \rightarrow \text{objects}
\end{align*}
\]

\[
\text{objects } \cap \text{arrows } = \emptyset
\]

The condition asserts that there is no intersection between the arrows and objects.
The non-terminal relationships of a graph are concerned with hyperarrows, their source, target and tag. The non-terminal relationships are formally defined in the schema NTGraph below. This schema is generic over the classes CComponents and CTag. It consists of the schema TGraph over CComponents, together with four additional components. The component hArrows is a finite sub-set of the 'components' component. The components hSource and hTarget are functions from hArrows to sequences of objects. The component hTag is a function from hArrows to the class CTag.

\[
\text{NTGraph}[\text{CComponents},\text{CTag}] \\
\text{TGraph[}\text{CComponents}] \\
\text{hArrows :}\{\text{F}\}\text{ components} \\
\text{hSource :}\text{hArrows} \rightarrow \text{seq objects} \\
\text{hTarget :}\text{hArrows} \rightarrow \text{seq objects} \\
\text{hTag :}\text{hArrows} \rightarrow \text{CTag} \\
\]

\[
\text{objects} \cap \text{hArrows} = \emptyset \\
\text{arrows} \cap \text{hArrows} = \emptyset \\
\text{objects} \cup \text{arrows} \cup \text{hArrows} = \text{components} \\
\]

Together with the condition of TGraph, the conditions assert that the sets: objects, arrows and hyperarrows, partition the components set.

In defining a class of actual NTGraph, it is necessary to define the actual generic parameters of the schema. These are presented in the following table:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Class Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>{c1, c2, \ldots}</td>
</tr>
<tr>
<td>Tag</td>
<td>any string</td>
</tr>
</tbody>
</table>

Any graph can now be represented as an element of the class

\[
\text{NTGraph[}\text{Component},\text{Tag}] \\
\]
Consider, for example, the following graph:

![Graph Diagram]

The following table illustrates how this graph is a member of the class:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>components</td>
<td>{c1,c2,c3,c4}</td>
</tr>
<tr>
<td>objects</td>
<td>{c1,c2}</td>
</tr>
<tr>
<td>arrows</td>
<td>{c3}</td>
</tr>
<tr>
<td>source</td>
<td>{(c3,c1)}</td>
</tr>
<tr>
<td>target</td>
<td>{(c3,c2)}</td>
</tr>
<tr>
<td>hArrows</td>
<td>{c4}</td>
</tr>
<tr>
<td>hSource</td>
<td>{(c4,&lt;c1&gt;)}</td>
</tr>
<tr>
<td>hTarget</td>
<td>{(c4,&lt;c2&gt;)}</td>
</tr>
<tr>
<td>hTag</td>
<td>{(c4,TTSPN)}</td>
</tr>
</tbody>
</table>
9.2.4 Picture Layout Grammars

The syntactic structures of graph grammars have, as basic components, a collection of picture elements. Formally, these components can be defined with a Z schema, TPictureComponents, generic over a class of lines CElement. The schema consists of a single attribute, elements which is a finite sub-set of CElement:

\[
\text{TPictureComponents}[\text{CElement}]
\]

\[
\text{elements} : \mathcal{P}\text{CElement}
\]

The schema has no conditions.

For present purposes, the terminal relationships of a picture are concerned with the type of the element and the over and left of relationships. The terminal relationships are formally defined in the schema TPicture below. This schema is generic over the class CElement and CElementType. It consists of the schema TPictureComponents over CElement, together with three additional components. The type component is a partial function from elements to CElementType. The over and leftOf components are relations between elements.

\[
\text{TPicture}[\text{CElement},\text{CElementType}]
\]

\[
\text{TPictureComponents}[\text{CElement}]
\]

\[
\text{type} : \text{elements} \mapsto \text{CElementType}
\]

\[
\text{over} : \text{elements} \leftrightarrow \text{elements}
\]

\[
\text{leftOf} : \text{elements} \leftrightarrow \text{elements}
\]

There is no condition.
The non-terminal relationship of a picture is a tag of a picture element. The non-terminal relationship is formally defined in the schema NTPicture below. This schema is generic over the classes CElement, CElementType and CTag. It consists of the schema TPicture over CElement and CElementType, together with an additional component tag. The component tag is a partial function from elements to the class CTag.

\[
\text{NTPicture} \left[ \text{CElement}, \text{CElementType}, \text{CTag} \right]
\]

\[
\text{TPicture} \left[ \text{CElement}, \text{CElementType} \right]
\]

\[
\text{tag} : \text{elements} \leftrightarrow \text{CTag}
\]

\[
\text{dom(type)} \cup \text{dom(tag)} = \text{elements}
\]

\[
\text{dom(type)} \cap \text{dom(tag)} = \emptyset
\]

The condition states that the domains of functions: type and tag, partition the elements set.

In defining a class of actual NTPicture, it is necessary to define the actual generic parameters of the schema. These parameters are presented in the following table:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Class Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>{e1, e2, .....}</td>
</tr>
<tr>
<td>ElementType</td>
<td>a set of strings, each of which can be interpreted as a picture element</td>
</tr>
<tr>
<td>Tag</td>
<td>any string that is not an element type</td>
</tr>
</tbody>
</table>

Any picture can now be represented as an element of the class

\[
\text{NTPicture} \left[ \text{Element}, \text{ElementType}, \text{Tag} \right]
\]
Consider, for example, the following picture represented in the functional notation:

\[ \text{over(Block, Tower)} \]

The following table illustrates how this graph is a member of the class:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>elements</td>
<td>{e_1, e_2}</td>
</tr>
<tr>
<td>type</td>
<td>{e_1, \text{Block}}</td>
</tr>
<tr>
<td>over</td>
<td>{(e_1, e_2)}</td>
</tr>
<tr>
<td>leftOF</td>
<td>{}</td>
</tr>
<tr>
<td>tag</td>
<td>{(e_2, \text{Tower})}</td>
</tr>
</tbody>
</table>
9.3 Syntactic Structure Classes

In the previous section, four classes of syntactic structure were defined using the Z notation. This section defines a generalised notion of syntactic structure. The syntactic structure classes outlined in the previous section are then employed to demonstrate the consistency and the completeness of the definition.

9.3.1 The Generalised Notion

For the generalised definition of syntactic structure, it will be necessary to have a class of names with which to name the relations of the syntactic structure. The class is introduced with the following basic type definition:

\[\text{[Names]}\]

The generalised notion of a structure class is now introduced with the StructureClass schema generic over the classes CStructures and Universe. The class consists of six components. The structures component is a subset of the class CStructures. It represents all the syntactic structures of the class. The 'components' component is a function from the class of structures to the subsets of the Universe. It represents the basic components of the structure. The tRNames component is a set of Names. It represents the names of the terminal relationships. The tRelations component assigns a relation between elements of the universe to structures and tRNames. The type is represented in curried form, that is, a function to a function, rather than cross product form to ease later notation. The tRelations component represents all the terminal relations of the structure class. The nTRNames component is a set of Names. It represents the names of the non-terminal relationships. The nTRelations component assigns a relation between elements of the universe to structures and nTRNames. Again the type is represented in curried form. The nTRelations component represents all the non-terminal relations of the structure class.

A simplification is made concerning the terminal and non-terminal relationships, in that they are all assumed to be binary. The purpose is to simplify the specifications that follow. It is a valid simplification, since n-ary relationships can always be decomposed into binary relationships.
StructureClass \{CStructures, Universe\}

\[
\begin{align*}
\text{structures} & : \mathcal{P} \text{CStructures} \\
\text{components} & : \text{structures} \rightarrow \mathcal{P} \text{Universe} \\
\text{tRNames} & : \mathcal{P} \text{Names} \\
\text{tRelation} & : \text{structures} \\
& \quad \rightarrow (\text{tRNames} \rightarrow (\text{Universe} \leftrightarrow \text{Universe})) \\
\text{nTRNames} & : \mathcal{P} \text{Names} \\
\text{nTRelation} & : \text{structures} \\
& \quad \rightarrow (\text{nTRNames} \rightarrow (\text{Universe} \leftrightarrow \text{Universe}))
\end{align*}
\]

\[
\text{tRNames} \cap \text{nTRNames} = \emptyset
\]

\[
\exists s : \text{structures} \cdot \\
\quad \text{components}(s) = \emptyset \\
\quad \land \quad \forall n : \text{tRNames} \cdot \text{tRelation}(s)(n) = \emptyset \\
\quad \land \quad \forall n : \text{nTRNames} \cdot \text{nTRelation}(s)(n) = \emptyset
\]

The first condition states that there is no intersection between the terminal and non-terminal relation names.

The second component asserts that there exists a structure for which the following hold:

- The structure has no components.
- All the terminal relations are empty
- All the non-terminal relations are empty.
The second condition ensures that each structure class has a null structure. There are a number of useful functions that can be defined upon StructureClass. These are:

nullStructure
terminalStructures
nonTerminalStructures

These functions are defined with generic axiomatic descriptions below.

The function nullStructure is a function from a StructureClass, generic over CStructures and Universe, to CStructures.

----=[CStructures,Universe]==================================

nullStructure : StructureClass[CStructures,Universe] → CStructures

---------------------------------------------------------------------

∀ S : StructureClass[CStructures,Universe] •
  nullStructure(M).components = Ø
  ∀ n : tRNames • tRelation(nullStructure(S))(n) = Ø
  ∀ n : nTRNames • nTRelation(nullStructure(S))(n) = Ø

The conditions state that null structure of a structure class satisfies the following:

- It has no components.
- All the terminal relations are empty
- All the non-terminal relations are empty.

The nullStructure is guaranteed to exist by the second condition of the StructureClass schema.

The functions terminalStructures and nonTerminalStructures are both total functions from StructureClass, generic over CStructures and Universe to a subset of CStructures.
The conditions state that:

- For any structure class, the terminal structures of that structure class are those structures with each non-terminal relation empty.
- For any structure class, the non-terminal structures are those that are not terminal.
9.3.2 Consistency and Completeness

In the previous sub-section the generalised notion of a syntactic structure class was developed by defining a class of all syntactic structure classes. The following sub-sections discuss the consistency and completeness of this generalised notion.

The generalised notion of a syntactic structure class bears a class-instance relationship with individual syntactic structure classes. Consistency and completeness can be viewed in terms of this class-instance distinction.

A definition is consistent if it does not contain contradictions. If the definition of the class of all syntactic structure classes is contradictory, then the class will have no instances. Thus, in order to demonstrate the consistency of the generalised notion, it is sufficient to demonstrate that there are instances of the class of all syntactic structure classes. The following four sections demonstrate how each of the syntactic structure classes defined in Section 9.2 are instances of the generalised notion of a syntactic structure class.

A definition can be considered complete if it covers all of the areas of concern. For the definition of the class of all syntactic structure classes to be complete, it is necessary to demonstrate that class incorporates all the syntactic structure classes that might be of interest in the design of formal languages, in a manner that simplifies the definition of the syntax of formal languages. It is impossible to demonstrate such consistency absolutely. The best that can be done is to demonstrate that:

- The generalised notion accounts for all existing syntactic structure classes.
- The generalised notation facilitates the definition of the syntax of formal languages, by enabling the definition of new syntactic structure classes.

The demonstration of the class-instance relationships between the generalised notion and the syntactic structure classes of Section 9.2 which occur in the next four sub-sections go a long way to demonstrating the first point. Section 9.6 of this chapter demonstrates how new syntactic structure classes can be
developed in order to create new grammatical formalisms. Chapter 10 demonstrates the development of a new grammatical formalism with the specific purpose of defining the syntax of an existing graphical notation.

The next four sections, then, demonstrate that a class-instance relationship holds between the generalised notion and syntactic structure classes of Section 9.2. This is done by demonstrating how the syntactic structure class is an instance of the class defined by the schema StructureClass defined in Sub-Section 9.3.1. Each demonstration takes the form of figures (see, for example, Figure 9.1).

The figure illustrates the six components of the StructureClass schema. The structures component is represented as a rectangle and the elements of the particular structure class which are instances of structures as round-cornered rectangles inside this box. The generic parameter Universe is represented as a rectangle and the elements of the particular structure class which are instances of the sub-sets of the Universe as round-cornered rectangles inside this box. The components function is represented as an unlabelled arrow, from the instances of structures to the instances of the Universe. Finally, both the terminal and non-terminal relationships, \( tRelation \) and \( nTRelation \), and their names, \( tRNames \) and \( nTRNames \), are represented with text-labelled arrows between the round-cornered rectangles inside the Universe. The arrows are drawn in the same manner as \( Z \) relationships, in order to aid understanding.

The final component of each sub-section describes the nullStructure of each structure class, as well as the terminal and non-terminal structures.
9.3.3 Text-based Grammars

The syntactic structure class of text-based grammars was formally described with schemas TStringComponents, TString and NTString in Sub-Section 9.2.1. This sub-section outlines how the syntactic structure class forms an instance of the schema StructureClass, as well as outlining the null structure and the terminal and non-terminal structures of the structure class.

Figure 9.1 demonstrates how schemas TStringComponents, TString and NTString give rise to instances of the six components of the schema StructureClass. The component positions of the schema TStringComponents is a sub-set of both the structures component of the StructureClass and of the generic parameter Universe. The other two members of the parameter Universe are the CAAlphabet and CNAAAlphabet parameters of the TString and NTString schemas. An unlabelled arrow represents the 'components' component of the Structure Class schema. The component after, next character and nTCharacter of the TString and NTString schemas are represented as text-labelled arrows. The names, 'after' and 'character ', thus form elements of the tRNames component of the StructureClass schema. The relationships denoted by 'after' and 'character' are the result of applying the tRelation component of the schema StructureClass. Similarly, the names, 'next' and 'nTCharacter', thus form elements of the nTRNames component of the StructureClass schema. The relationships denoted by 'next' and 'nTCharacter' are the result of applying the nTRelation component of the schema StructureClass.

The nullStructure is the string with no positions. In other words, the empty string. The terminalStructures are the strings without a non-terminal character. All other strings are non-terminal structures.
Figure 9.1 The Text-Based Structure Class as an Instance of the schema StructureClass
9.3.4 Shape Grammars

The syntactic structure class of shape grammars was formally described with schemas TShapeComponents, TShape and NTShape in Sub-Section 9.2.2. This sub-section outlines how this syntactic structure class forms an instance of the schema StructureClass, as well as outlining the null structure and the terminal and non-terminal structures of the structure class.

Figure 9.2 demonstrates how schemas TShapeComponents, TShape and NTShape give rise to instances of the six components of the schema StructureClass. The component lines, of the schema TShapeComponents, is a sub-set of both the structures component of the StructureClass and the generic parameter Universe. The other members of the parameter Universe are the sets Reals, \( \mathbb{P}(\text{Reals} \times \text{Reals}) \), Lines and CTag from the TShape and NTShape schemas. An unlabelled arrow represents the 'components' component of the Structure Class schema. The component hStart, hEnd, vStart, vEnd, values, tag of the TShape and NTShape schemas are represented as text labelled arrows. The names, 'hStart', 'hEnd', 'vStart', 'vEnd' and 'values' thus form elements of the tRNames component of the StructureClass schema. The relationships denoted by 'hStart', 'hEnd', 'vStart', 'vEnd' and 'values' are the result of applying the tRelation component of the schema StructureClass. Similarly, the name 'tag' thus forms an element of the nTRNames component of the StructureClass schema. The relationship denoted by 'tag' are the result of applying the nTRelation component of the schema StructureClass.

In general, mapping the, (not necessarily binary), relationships of structure class descriptions, onto the, (necessarily), binary relationships of the schema StructureClass, may give rise to extra relationships. In the shape grammar example an extra \( \epsilon \) arrow connects the Universe elements lines and \( \mathbb{P} \) lines illustrating such a situation. A more complex set of relationships exists between Universe elements Reals and \( \mathbb{P}(\text{Reals} \times \text{Reals}) \). These relationships are not presented for reasons of clarity.

The nullStructure is the shape with no lines. In other words, the empty shape. The terminalStructures are the shapes without tags. All other shapes are non-terminal structures.
structures

lines

Universe

\( P \) lines

\( \in \)

CTag

\( P(\text{Reals} \times \text{Reals}) \)

hStart

hEnd

vStart

vEnd

Reals

Figure 9.2 The Shape Structure Class as an Instance of the schema StructureClass
9.3.5 Graph Grammars

The syntactic structure class of graph grammars was formally described with schemas TGraphComponents, TGraph and NTGraph in Sub-Section 9.2.3. This sub-section outlines how this syntactic structure class forms an instance of the schema StructureClass, as well as outlining the null structure and the terminal and non-terminal structures of the structure class.

Figure 9.3 demonstrates how schemas TGraphComponents, TGraph and NTGraph give rise to instances of the six components of the schema StructureClass. The component 'components', of the schema TGraphComponents, is a sub-set of both the structures component of the StructureClass and the generic parameter Universe. The other members of the parameter Universe are the sets objects, arrows, hArrows, seq objects and CTag from the TGraph and NTGraph schemas. An unlabelled arrow represents the 'components' component of the StructureClass schema. The components objects, arrows, source, target, hArrows, hSource, hTarget and hTag of the TGraph and NTGraph schemas are represented as text-labelled arrows. The names, 'objects', 'arrows', 'source' and 'target' thus form elements of the tRNames component of the StructureClass schema. The relationships denoted by 'objects', 'arrows', 'source' and 'target' are the result of applying the tRelation component of the schema StructureClass. Similarly, the names, 'hArrows', 'hSource', 'hTarget' and 'hTag' thus form elements of the nTNames component of the StructureClass schema. The relationship denoted by 'hArrows', 'hSource', 'hTarget' and 'hTag' are the result of applying the nTRelation component of the schema StructureClass.

The null structure is the graph with no components. In other words, the empty graph. The terminal structures are the graphs without any hyperArrows, since this forces all non-terminal relationships to be empty. All other graphs are non-terminal structures.
Figure 9.3 The Graph Structure Class as an Instance of the schema StructureClass
9.3.6 Picture Layout Grammars

The syntactic structure class of picture layout grammars was formally described with schemas TPictureComponents, TPicture and NTPicture in Sub-Section 9.2.3. This sub-section outlines how this syntactic structure class forms an instance of the schema StructureClass, as well as outlining the null structure and the terminal and non-terminal structures of the structure class.

Figure 9.4 demonstrates how schemas TPictureComponents, TPicture and NTPicture give rise to instances of the six components of the schema StructureClass. The component 'elements', of the schema TPictureComponents, is a sub-set of both the structures component of the StructureClass and the generic parameter Universe. The other members of the parameter Universe are the sets CElementType and CTag from the TPicture and NTPicture schemas. An unlabelled arrow represents the 'components' component of the Structure Class schema. The components type, over, leftOf and tag of the TPicture and NTPicture schemas are represented as text-labelled arrows. The names, 'type', 'over' and 'leftOf' thus form elements of the tRNames component of the StructureClass schema. The relationships denoted by 'type', 'over' and 'leftOf' are the result of applying the tRelation component of the schema StructureClass. Similarly, the name 'tag' thus forms an element of the nTRNames component of the StructureClass schema. The relationship denoted by 'tag' is the result of applying the nTRelation component of the schema StructureClass.

The null structure is the picture with no elements. In other words, the empty picture. The terminal structures are the pictures without a tag. All other shapes are non-terminal structures.
9.3.7 Conclusion

This section has defined the generalised notion of a syntactic structure class with the schema StructureClass. The consistency of this definition has been demonstrated by showing that each of the four syntactic structure classes specified in Section 9.2 can be viewed as instances of the class defined by the StructureClass schema. Since the choice of syntactic structure classes specified in Section 9.2 forms a broad sample of all those in existence, evidence has also been provided of the completeness of the generalised notion of a syntactic structure class. Further evidence of completeness is presented in Section 9.6 and Chapter 10.
9.4 Terminal And Non-terminal Categories

This section introduces the terminal and non-terminal categories that are used to develop the generalised notion of derivation. Prior to introducing the categories, the notions of terminal and non-terminal arrows are introduced.

A terminal arrow is a relationship between the basic components of syntactic structures that preserves terminal relationships. The concept of a terminal arrow is defined with the schema TArrow generic over the classes CStructures and Universe. It has six components. The structClass component is a structure class generic over the classes CStructures and Universe. The source and target components are structures of the structClass. The basic arrow is a relation between the components of the source and the components of the target. The arrow is a relation on the universe.

\[
\text{TArrow \in [\text{CStructures, Universe}]}
\]

\[
\begin{align*}
\text{structClass} & : \text{StructureClass[CStructures, Universe]} \\
\text{source} & : \text{structClass.structures} \\
\text{target} & : \text{structClass.structures} \\
\text{basicArrow} & : \text{structClass.components(source)} \\
& \quad \leftrightarrow \text{structClass.components(target)} \\
\text{arrow} & : \text{Universe} \leftrightarrow \text{Universe}
\end{align*}
\]

\[
\forall sc, tc : \text{Universe} \cdot \\
(sc, tc) \in \text{arrow} \\
\supset \\
((sc, tc) \in \text{basicArrow} \lor (sc \in \text{dom(basicArrow)} \land sc = tc))
\]

\[
\forall sc1, sc2, tc1, tc2 : \text{Universe} \cdot \\
((sc1, tc1) \in \text{arrow} \land (sc2, tc2) \in \text{arrow}) \\
\supset \\
\exists n : \text{tRNames} \cdot \\
(sc1, sc2) \in \text{tRelation(source, n)} \\
\supset \\
(tc1, tc2) \in \text{tRelation(target, n)}
\]
The conditions state:

- The arrow relation is an extension of the basic arrow relation to the universe, so identity mappings are added for each element not in the domain of basicArrow.
- The arrow relation preserves the terminal relationships.

To make the specifications that follow easier to understand, a function tArrows is defined using the generic form of an axiomatic description. Applied to a structure class, the function returns all the terminal arrows of that structure class. The function tArrows maps a StructureClass, generic over the classes CStructures and Universe, to a set of TArrows, generic over the classes CStructures and Universe.

\[
\text{tArrows} : \text{StructureClass}[\text{CStructures,Universe}] \rightarrow \mathcal{P} \text{TArrow}[\text{CStructures,Universe}]
\]

\[
\forall S : \text{StructureClass}[\text{CStructures,Universe}] \cdot \text{tArrows}(S) = \{A : \text{TArrow}[\text{CStructures,Universe}] \mid A.\text{structClass} = S\}
\]

The condition states, that for all structure classes, the tArrows of the structure class is all the TArrows with that structure class.

A non-terminal arrow is an injective function between the basic components of syntactic structures that preserves non-terminal relationships. The concept of a non-terminal arrow is defined with the schema NTArrow generic over the classes CStructures and Universe. It consists of a the class TArrow generic over the classes CStructures and Universe.
The conditions state that:

- The basic arrow is an injective function from the components of the source to the components of the target.
- The arrow relation preserves the non-terminal relationships.

Since NTArrow is a specialisation of TArrow, it preserves both terminal and non-terminal arrows.

As for terminal arrows, to make the specifications that follow easier to understand, a function nTArrows is defined using the generic form of an axiomatic description. Applied to a structure class, the function returns all the non-terminal arrows of that structure class. The function nTArrows maps a StructureClass, generic over the classes CStructures and Universe, to a set of NTArrows, generic over the classes CStructures and Universe.
The condition states that for all structure classes, the tArrows of the structure class is all the TArrows with that structure class.

The non-terminal arrows of a structure class is a sub-set of the terminal arrows of that structure class.

Having defined the concepts of terminal and non-terminal arrows, it is now possible to define the terminal and non-terminal categories that are based upon these arrows.

The terminal category of a structure class is a category whose objects are syntactic structures and arrows are terminal arrows. A generic axiomatic description is used to define the function tCategory which maps StructureClass, generic over the classes CStructures and Universe, to the class Category generic over CStructures and the class of TArrows, which, is in turn, generic over the classes CStructures and Universe.
tCategory : StructureClass[CStructures,Universe] → Category[
    CStructures,
    TArrows[CStructures,Universe]]

∀ S : StructureClass[CStructures,Universe] •
    tCategory(S).objects = S.structures
∧ tCategory(S).arrows = tArrows(S)
∧ ∀ f : tCategory(S).arrows •
    tCategory(S).source(f) = f.source
    ∧ tCategory(S).target(f) = f.target
∧ ∀ o: tCategory(S).objects •
    tCategory(S).id(o).source = o
    ∧ tCategory(S).id(o).target = o
    ∧ tCategory(S).id(o).arrow = \{ x,y : Universe \mid x = y \}
∧ ∀ f,g : tCategory(S).arrows •
    g°tCategory(S)f .source = f.source
    ∧ g°tCategory(S)f .target= g.target
    ∧ g°tCategory(S)f .arrow = g.\arrow°f .arrow

The condition states that for every structure class:

- The objects of the tCategory are the structures.
- The arrows are the terminal arrows.
- The source and target functions of the category are given by the source and target components of the arrow.
- The identity arrow is the identity relation between source and target.
- The composition of arrows is given by the composition of the arrow component relation.

It can be seen that tCategory is indeed a category.
The non-terminal category of a structure class is a category whose objects are syntactic structures and arrows are non-terminal arrows. A generic axiomatic description is used to define the function nTCategory which maps StructureClass, generic over the classes CStructures and Universe, to the class Category generic over CStructures and the class of NTArrows, which, is in turn, generic over the classes CStructures and Universe.

nTCategory : StructureClass[CStructures,Universe] → Category[
                           CStructures,
                           NTArrows[CStructures,Universe]]

∀ S : StructureClass[CStructures,Universe] •
   nTCategory(S).objects = S.structures
   ∧ nTCategory(S).arrows = nTArrows(S)
   ∧ nTCategory(S).source
       = nTArrows(S) ◁ tCategory(S).source
   ∧ nTCategory(S).target
       = nTArrows(S) ◁ tCategory(S).target
   ∧ nTCategory(S).id = tCategory(S).id
   ∧ nTCategory(S).¬°
       = nTArrows(S) ◁ tCategory(S).¬°
The condition states that for every structure class:

- The objects of the tCategory are the structures.
- The arrows are the non-terminal arrows
- The source function of the category is a restriction of the source function of the terminal category to the non-terminal arrows.
- The target function of the category is a restriction of the target function of the terminal category to the non-terminal arrows.
- The identity function of the category is a restriction of the identity function of the terminal category
- The composition function of the category is a restriction of the composition function of the terminal category to the non-terminal arrows.

It can be seen that nTCategory is indeed a category. In fact, it is a subcategory of tCategory.

The final specification in this section distinguishes a special set of non-terminal structures. These structures are basic in the sense that they have no non-null sub-structures. It is precisely this class that will form the 'left-hand side' or source of the production rules. The basic non-terminals amount to the generalisation of a single terminal in string grammars. A generic axiomatic description is used to define the basicNonTerminals as a function, from the StructureClass generic over the classes CStructures and Universe to a set of CStructures.
The condition states that for all structure classes:

- The basic non-terminals is a subset of the non-terminals
- A non-terminal structure is basic if the only non-terminal arrow whose target is that structure comes from the null structure or the structure itself.
9.5 Grammar and Derivation

The concepts of terminal and non-terminal categories developed in the previous section are used in this section to construct a generalised notion of grammar and derivation. The generalised notions are partially verified by demonstrating derivations for each of the grammars considered in Sections 1 and 2.

9.5.1 The General Notions of Grammars and Derivation

The concept of a grammar can be formally described using a Z schema. The aim is to describe context-free grammars, those for which the source of the production rule is a basic non-terminal. The schema Grammar is generic over the classes CStructures and Universe. It has three components. The structClass is the structure class upon which the grammar is based. The startSymbol is a basic non-terminal of the structure class. The productionRules is a set of tArrows (terminal arrows) of the structure class.

\[
\text{Grammar} \left[ \text{CStructures, Universe} \right]
\]

\[
\begin{align*}
\text{structClass} & : \text{StructureClass}[\text{CStructures, Universe}] \\
\text{startSymbol} & : \text{basicNonTerminals}(\text{structClass}) \\
\text{productionRules} & : \left[ P \right. \text{tArrows}(\text{structClass})
\end{align*}
\]

\[
\forall p : \text{productionRules} \cdot \\
p.\text{source} \in \text{basicNonTerminals}(\text{structClass})
\]

The condition states that the source of each production rule is a basic non-terminal.

Having described the notion of a grammar, the next step is to define the notion of a derivation. In any grammar, a derivation is based upon some production rule. The source of the production must be matched with the target of the production. The source of the production rule will typically
match only part of the source of the derivation. This part will be called the old. The part of the production that does not match the source of the production rule will be called the not matched. Finally, the part of the target of the derivation that matches the target of the production rule will be called the new.

The concept of a derivation can be defined using a Z schema generic over the classes CStructures and Universe. It consists of a Grammar over the generic classes, together with seven additional components. The rule component is a production rule. The derivation component is a terminal arrow. The sourceMatch and targetMatch components are non-terminal arrows. The components, old and notMatched are sub-sets of the structure class components of the source of the derivation. The component new is a sub-set of the structure class components of the target of the derivation.

The conditions state that:

- The sourceMatch and derivation arrows commute with the production rule and the targetMatch arrows in the terminal category of the structure class.
- The set old is the range of the sourceMatch arrow.
- The set new is the range of the targetMatch arrow.
- The set nonMatched is the set of all the components in the source of the derivation that are not in the set old.
- There is no intersection between the sets notMatched and new.
- The components of the target of the derivation is the union of the sets new and notMatched.
- The derivation arrow consists of the components of the identity arrow restricted to the set notMatched, together with all the ordered pairs whose first component is an element of the set old and whose second component is an element of the set new, such that, there is some component of the source of the production rule, which is mapped to the old component by the sourceMatch, and mapped to the new component by the composition of the rule and the targetMatch.
Derivation [CStructures, Universe]

Grammar [CStructures, Universe]

rule : productionRules

 derivation : tArrows(structClass)

 sourceMatch,

 targetMatch : nTArrows(structClass)

 old,

 notMatched : \text{structClass}.components(derivation.source)

 new : \text{structClass}.components(derivation.target)

 sourceMatch

 derivation

 rule

 targetMatch

 commutes in tArrow(structClass)

 old = \text{ran}(sourceMatch.arrow)

 new = \text{ran}(targetMatch.arrow)

 notMatched

 = \text{structClass}.components(derivation.source) \setminus old

 notMatched \cap new = \emptyset

 \text{structClass}.components(derivation.target)

 = notMatched \cup new

 derivation.arrow =

 notMatched \leftarrow \text{nTCategory(structClass).id}(derivation.source)

 \cup

 \{o:old; n:new \ |

 \exists c : \text{structClass}.components(\text{dom}(sourceMatch)) \bullet

 (c,o) \in sourceMatch.arrow

 \land

 (c,n) \in (targetMatch \circ \text{Category(structClass).rule}).arrow

 300
The conditions describe how a production rule is applied to a syntactic structure to produce a new syntactic structure through a derivation. The first condition demonstrates the situation of the four arrows graphically and illustrates how they commute. The next three conditions formally describe the elements of the sets old, new, and notMatched. The next two conditions demonstrate how the components of the target of the derivation consist of the disjoint union of the notMatched components and the new components. The final condition illustrates how the components of the derivation arrow are determined. It consists of the identity function on the notMatched components, together with a set of ordered pairs derived from the sourceMatch and the production rule.

The aim of a grammar is to define a language. In order to develop a formal notion of the language defined by a grammar, it will be useful to develop the notion of a derivation category of a grammar. In turn, the development of the notion of a derivation category, requires the notion of the derivation arrows of a grammar.

The concept of the derivations of a grammar can be defined using an axiomatic description generic over the classes CStructures and Universe. The derivationArrows is a function from a grammar over the generics to the sets of arrows over the generics.

The condition states that for any grammar, the derivation arrows of that grammar are all the terminal arrows of that structure class of the grammar that are either:

- Identity arrows.
- The derivation arrows for some derivation of the grammar.
- Compositions of other derivation arrows.
Having defined the derivation arrows, it is possible to define the derivation category of a grammar using an axiomatic description generic over the classes CStructures and Universe. The derivationCategory is a function from a grammar over the generics to a category over the CStructures and the terminal arrows of the generics.
The condition states that for all grammars:

- The objects of the category are the structures of the structure class of the grammar.
- The arrows are the derivation arrows of the grammar.
- The source function of the category is a restriction of the source function of the terminal category to the derivation arrows.
- The target function of the category is a restriction of the target function of the terminal category to the non-terminal arrows.
- The identity function of the category is the identity function of the terminal category.
- The composition function of the category is a restriction of the composition function of the terminal category to the derivation arrows.

\[ \text{derivationCategory : Grammar[CStructures,Universe]} \rightarrow \text{Category[}
\begin{align*}
\text{CStructures} \\
\text{TArrow[CStructures,Universe]} \end{align*}\] 

\[ \forall G : \text{Grammar[CStructures,Universe]} \bullet \\
\begin{align*}
\text{derivationCategory.objects} &= G.\text{structClass.structures} \\
\text{derivationCategory.arrows} &= \text{derivationArrows}(G) \\
\text{derivationCategory(M).source} &= \text{derivationArrows}(M) \bowtie \text{tCategory(M).source} \\
\text{derivationCategory(M).target} &= \text{derivationArrows}(M) \bowtie \text{tCategory(M).target} \\
\text{derivationCategory(M).id} &= \text{tCategory(M).id} \\
\text{derivationCategory(M)}^{-\circ} &= \text{derivationArrows}(M) \bowtie \text{tCategory(M)}^{-\circ} \\
\end{align*}\]
Finally, it is possible to describe the language defined by a grammar using the axiomatic description generic over the classes CStructures and Universe. The languageDefinedBy is a function from a grammar over the generics to the CStructures.

The condition states that for any grammar, the language defined by the grammar is the set of terminal structures of the structure class of the grammar, such that there is an arrow in the derivation category whose target is the terminal structure and whose source is the start symbol.

$$\forall G : \text{Grammar[CStructures, Universe]} \cdot$$
$$\text{languageDefinedBy}(G) = \{ S : \text{terminalStructures}(G.\text{structClass}) \mid$$
$$\exists f : \text{derivationCategory}(G).\text{arrows}$$
$$f.\text{source} = G.\text{startSymbol} \land f.\text{target} = S\}$$

This schema completes the formal description of the generalised notion of a grammar. The next five sub-sections attempt to validate the notion of derivation using the grammars outlined in Sections 10.1 and 10.2.
9.5.2 Consistency and Completeness

In the previous sub-section the generalised notions of grammar and derivation were developed. The following sub-sections discuss the consistency and completeness of these generalised notions.

In Sub-section 9.3.2, evidence for the consistence and completeness of the generalised notion of syntactic structure class was presented by demonstrating how each of the syntactic structure classes, presented initially in Section 9.1 and formally defined in Section 9.2, formed an instance of the class defined by the schema StructureClass. In order to develop a similar demonstration of the consistency and completeness of the generalised notions of grammar and derivation, it would be necessary to formalise the definitions of grammar and derivation for each of the grammars presented in Section 10.1. Such an extensive formalisation is beyond the scope of this thesis.

The evidence for the completeness and consistency of the generalised notions of grammar and derivation presented here relates the general notions to the particular derivations presented in Section 9.1. In the four sub-sections that follow, it is illustrated how the general notions of grammar and derivation give rise to derivations for each of the four syntactic structure classes defined in Section 9.2. Each of these examples provides evidence of the consistency of the definitions of grammar derivation. Taken together they provide some evidence of the completeness of the definition.

Each of the following four sections presents a derivation. The explicit presentation of a derivation involves a great deal of detail. The Sub-sections 9.5.3 and 9.5.5 present the derivations explicitly, outlining the individual syntactic structures, production rules and derivations. Sub-sections 9.5.4 and 9.5.6 present only the derivations.
9.5.3 Text-based Grammars

This sub-section provides evidence of the consistency and completeness of the generalised notion of grammar by presenting an example derivation for the text-based case. The sub-section demonstrates how the string 1DB can be derived from the string 1B with the production rule:

\[
B \rightarrow DB
\]

The first requirement is to present the four strings according to the formalisation of the text-based syntactic structure class from Sub-section 9.2. As in 9.2, the strings will be represented in tables. In addition, in order to present a derivation, they will also be represented graphically. The graphical representations of the strings may be thought of instances of the graphical representation of the text-based structure class of Figure 9.1.

The string B is represented in table form as follows:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>positions</td>
<td>{p1}</td>
</tr>
<tr>
<td><em>after</em></td>
<td>{}</td>
</tr>
<tr>
<td>character</td>
<td>{}</td>
</tr>
<tr>
<td>next</td>
<td>{}</td>
</tr>
<tr>
<td>nTCharacter</td>
<td>{(p1,B)}</td>
</tr>
</tbody>
</table>

The string B can be represented graphically as follows:

```
  B
   ▲
   | nTCharacter
   |   ▲
   |   | p1
   |   v
   v
```
The string DB is represented in table form as follows:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>positions</td>
<td>{p1,p2}</td>
</tr>
<tr>
<td><em>after</em></td>
<td>{((p2,p1))}</td>
</tr>
<tr>
<td>character</td>
<td>{}</td>
</tr>
<tr>
<td>next</td>
<td>{((p1,p2))}</td>
</tr>
<tr>
<td>nTCharacter</td>
<td>{((p1,D),(p2,B))}</td>
</tr>
</tbody>
</table>

The string DB is represented graphically as follows:

```
  p1  after  p2
    |       |
    v       v
nTCharacter  next  nTCharacter
```

D   B

The string IB is represented in table form as follows:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>positions</td>
<td>{p1,p2}</td>
</tr>
<tr>
<td><em>after</em></td>
<td>{((p2,p1))}</td>
</tr>
<tr>
<td>character</td>
<td>{((p1,1))}</td>
</tr>
<tr>
<td>next</td>
<td>{((p1,p2))}</td>
</tr>
<tr>
<td>nTCharacter</td>
<td>{((p2,B))}</td>
</tr>
</tbody>
</table>

The string IB is represented graphically as follows:

```
  1  after  B
    |       |
    v       v
p1     after     p2
    |       |
    v       v
character  nTCharacter
```


The string 1DB is represented in table form as follows:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>positions</td>
<td>{p1,p2,p3}</td>
</tr>
<tr>
<td><em>after</em></td>
<td>{(p2,p1), (p3,p1),(p3,p2)}</td>
</tr>
<tr>
<td>character</td>
<td>{(p1,1)}</td>
</tr>
<tr>
<td>next</td>
<td>{(p1,p2),(p2,p3)}</td>
</tr>
<tr>
<td>nTCharacter</td>
<td>{(p2,D),(p3,B)}</td>
</tr>
</tbody>
</table>

The string 1DB is represented graphically as follows:

Having represented each of the strings, it is now possible to represent the production rule.

\[ B \rightarrow DB \]

Recall that a production rule is a terminal arrow and that a terminal arrow has five components:

- structure class (in this case the class NTString)
- source (in this case the string B)
- target (in this case the string DB)
- basicArrow
- arrow
For all terminal arrows, and indeed, non-terminal arrows, the arrow is determined by the basic arrow. Thus, to represent the production rule graphically, it is necessary to present the two strings and the relationships of the basic arrow. This is done in figure 9.5:

![Diagram of production rule B -> DB](image)

Figure 9.5 A Graphical Representation of the Production Rule B -> DB

The ordered pairs of the basic arrow relationship are represented with greyed lines. To ensure that Figure 9.5 does indeed represent a terminal arrow it is necessary to ensure that all the terminal relations of the source string are preserved. In this case, there are no terminal relationships in the source string and hence they are all preserved.

Having represented the production rule, the next step is to represent the derivation. Using the same greyed arrow approach, the whole derivation is represented in Figure 9.6
Figure 9.6 The Graphical Representation of the Derivation of 1DB from 1B using Production Rule B -> DB

It can be demonstrated that this is indeed a derivation according to the formal definition of the previous sub-section. If the strings in Figure 9.6 are presented in their usual form, making the terminal and non-terminal relationships implicit, then the representation of the derivation can be simplified, as in Figure 9.7.
Figure 9.7 The Simplified Graphical Representation of the Derivation of 1DB from 1B using Production Rule B → DB

Such pictures of derivations can be strung together, beginning with the start symbol, to indicate that a particular string is part of the language defined by the grammar. Figure 9.8 overleaf demonstrates that the string 01 can be derived using the string grammar given in Section 9.1.
Figure 9.8 A Demonstration that the String 01 is a part of the Language Defined by the Text-based grammar of Sub-section 9.1.1.
9.5.4 Shape Grammars

This sub-section provides evidence of the consistency and completeness of the generalised notion of grammar by presenting an example derivation for the shape grammar case. The sub-section demonstrates how the shape:

\[
\begin{array}{c}
  \begin{array}{c}
    \text{can be derived from the shape}
  \end{array}
  \end{array}
\]

with the production rule:

\[
\begin{array}{c}
  \begin{array}{c}
    \text{This production rule is an instance of the second production rule of section 10.1.2. For the purposes of the generalisation, those production rules are more easily considered as representing production rule schemas. That is, each of the original production rules represents a set of production rules. Each rule of the set preserves the size and position of the triangle, but the initial triangle may be of any size in any position.}
  \end{array}
  \end{array}
\]

In this case rather than present all the tables and diagrams representing the terminal and non-terminal relationships, culminating in a complicated picture of a derivation, the simplified form of a derivation is presented directly in Figure 9.9.
Each arrow represents three ordered pairs linking the different triangles. It can be demonstrated that Figure 9.9 does indeed represent a derivation, according to the formal definition of the previous sub-section, where the structure class is NTShapes.
9.5.5 Graph Grammars

This sub-section provides evidence of the consistency and completeness of the generalised notion of grammar by presenting an example derivation for the graph grammar case. It shall be demonstrated how the graph:

- - -
  • •

can be derived from the graph:

\[ \text{TTSPN} \]

- - -
  \( B1 \) \( E1 \)

with the production rule:

\[ \text{TTSPN} \rightarrow \cdot \cdot \]

The first requirement is to present the four graphs according to the formalisation of the graph structures from Section 9.2. As in 9.2, the graphs will be represented in tables. In addition, in order to present a derivation, they will also be represented graphically. The additional graphical representations of the graphs may be thought of as instances of the graphical representation of the graph structure class of Figure 9.3.
The graph

is represented in table form:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>components</td>
<td>{c1,c2,c3}</td>
</tr>
<tr>
<td>objects</td>
<td>{c1,c2}</td>
</tr>
<tr>
<td>arrows</td>
<td>{c3}</td>
</tr>
<tr>
<td>source</td>
<td>{(c3,c1)}</td>
</tr>
<tr>
<td>target</td>
<td>{(c3,c2)}</td>
</tr>
<tr>
<td>hArrows</td>
<td>{}</td>
</tr>
<tr>
<td>hSource</td>
<td>{}</td>
</tr>
<tr>
<td>hTarget</td>
<td>{}</td>
</tr>
<tr>
<td>hTag</td>
<td>{}</td>
</tr>
</tbody>
</table>

The same graph is represented graphically as follows:

![Graphical representation of the same graph](image)
The graph

\[
\text{TTSPN}
\]

is represented in table form:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>components</td>
<td>{c1,c2,c3}</td>
</tr>
<tr>
<td>objects</td>
<td>{c1,c2}</td>
</tr>
<tr>
<td>arrows</td>
<td>{}</td>
</tr>
<tr>
<td>source</td>
<td>{}</td>
</tr>
<tr>
<td>target</td>
<td>{}</td>
</tr>
<tr>
<td>hArrows</td>
<td>{c3}</td>
</tr>
<tr>
<td>hSource</td>
<td>{(c3,&lt;c1&gt;)}</td>
</tr>
<tr>
<td>hTarget</td>
<td>{(c3,&lt;c2&gt;)}</td>
</tr>
<tr>
<td>hTag</td>
<td>{(c3,TTSPN)}</td>
</tr>
</tbody>
</table>
The same graph is represented graphically as follows:

The graph

is represented in table form:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>components</td>
<td>{c1,c2,c3,c4}</td>
</tr>
<tr>
<td>objects</td>
<td>{c1,c2}</td>
</tr>
<tr>
<td>arrows</td>
<td>{c3}</td>
</tr>
<tr>
<td>source</td>
<td>{(c3,c1)}</td>
</tr>
<tr>
<td>target</td>
<td>{(c3,c2)}</td>
</tr>
<tr>
<td>hArrows</td>
<td>{c4}</td>
</tr>
<tr>
<td>hSource</td>
<td>{(c4,&lt;c1&gt;)}</td>
</tr>
<tr>
<td>hTarget</td>
<td>{(c4,&lt;c2&gt;)}</td>
</tr>
<tr>
<td>hTag</td>
<td>{(c4,TTSPN)}</td>
</tr>
</tbody>
</table>
The same graph is represented graphically as follows:

```
\[
\begin{array}{c}
\text{objects} \\
c_1 & c_2
\end{array}
\]\n```

```
\[
\begin{array}{c}
\text{source} \\
c_3 & c_4
\end{array}
\]\n```

```
\[
\begin{array}{c}
\text{target} \\
c_3 & c_4
\end{array}
\]\n```

```
\[
\begin{array}{c}
\text{arrows} \\
c_3 & c_4
\end{array}
\]\n```

Finally the graph

```
\[
\begin{array}{c}
\text{source} \\
& c_3
\end{array}
\]\n```

```
\[
\begin{array}{c}
\text{target} \\
& c_4
\end{array}
\]\n```

is represented in table form as follows:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>components</td>
<td>{c_1, c_2, c_3, c_4}</td>
</tr>
<tr>
<td>objects</td>
<td>{c_1, c_2}</td>
</tr>
<tr>
<td>arrows</td>
<td>{c_3, c_4}</td>
</tr>
<tr>
<td>source</td>
<td>{(c_3, c_1), (c_4, c_1)}</td>
</tr>
<tr>
<td>target</td>
<td>{(c_3, c_2), (c_4, c_2)}</td>
</tr>
<tr>
<td>hArrows</td>
<td>{}</td>
</tr>
<tr>
<td>hSource</td>
<td>{}</td>
</tr>
<tr>
<td>hTarget</td>
<td>{}</td>
</tr>
<tr>
<td>hTag</td>
<td>{}</td>
</tr>
</tbody>
</table>

The same graph is represented graphically as follows:

```
\[
\begin{array}{c}
\text{objects} \\
c_1 & c_2
\end{array}
\]\n```

```
\[
\begin{array}{c}
\text{source} \\
c_3 & c_4
\end{array}
\]\n```

```
\[
\begin{array}{c}
\text{target} \\
c_3 & c_4
\end{array}
\]\n```

```
\[
\begin{array}{c}
\text{arrows} \\
c_3 & c_4
\end{array}
\]\n```
Having represented each of the graphs, it is now possible to represent the production rule.

![Diagram of production rule](image)

The production rule, Figure 9.10, can be presented in the same manner as was done for strings, using grey arrows to indicate the terminal arrow. I

![Diagram of derivation](image)

**Figure 9.10** The Graphical Representation of a Graph Grammar Production Rule.

Having represented the production rule, the next step is to represent the derivation as a whole. Using the same greyed arrow approach, the whole derivation is represented in Figure 9.11.
Figure 9.11 The Graphical Representation of a Graph Grammar Derivation.

It can be demonstrated that this is indeed a derivation according to the formal definition of the previous sub-section. If the graphs of Figure 9.11 are represented in their usual form, then the representation of the derivation can be simplified, as in Figure 9.12.
Figure 9.12 The Simplified Graphical Representation of a Graph Grammar Derivation.
9.5.6 Picture layout Grammars

This sub-section provides evidence of the consistency and completeness of the generalised notion of grammar by presenting an example derivation for the picture layout grammar case. It shall be demonstrated how the picture:

\[
\text{over(\text{Block,Block})}
\]

which represents

\[
\quad \\
\quad \\
\]

\[
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\quad \\
\qua
It can be demonstrated that Figure 9.13 does indeed represent a derivation according to the formal definition of the previous sub-section, where the structure class is NTShapes.

9.5.7 Conclusion

This section has defined the generalised notions of a grammar and derivation. Evidence for the consistency of these definitions has been presented by illustrating how they account for the derivations for each of the four grammars presented in Section 9.1. Since these grammars form a broad sample of all those in existence, evidence has also been provided of the completeness of the generalised notions. Further evidence of completeness is presented in Section 9.6 and Chapter 10, where derivations are presented for two further grammatical formalisms.
9.6 Venn grammars

So far, this chapter has developed a generalised notion of grammatical formalism and demonstrated that four existing grammatical formalisms form instances of the generalised notion. The purpose of the generalised notion is to aid the specification of classes of formal languages. This section employs the generalised notion to develop a simple new grammatical formalism for defining languages whose terminal syntactic structures are Venn diagrams.

A Venn diagram consists of a collection of blobs, such that some blobs are enclosed within others. Traditionally the blobs represent sets. An example of a Venn diagram is as follows:

![Venn Diagram Example]

For notational convenience the Venn diagrams here are not drawn in the usual fashion, but follow the Higraph notation of Harel [Harel 88]. In set theoretic terms, following Harel means that the set E represents exactly the intersection of the sets A and B, and the union of the disjoint sets C, D and E is equal to the set A. Strictly speaking, the labels (A,B etc.) are unnecessary. They are used solely to enable the referencing of the diagrams from the text.

If the generalised notion of grammatical formalism is employed, defining a novel grammatical formalism requires only the definition of a syntactic structure class which is an instance of the schema structure class. This is done in the same manner as the syntactic structure classes of Section 9.2.
The basic components of a Venn diagram are blobs. They can be described with a schema generic over the class CBlob. It has a single component blobs, which is a sub-set of the class CBlob.

\[
\text{TVennDiagramComponents}[\text{CBlob}]
\]

\[
\begin{align*}
\text{blobs} & : \mathcal{P} \text{CBlob} \\
\end{align*}
\]

The schema has no conditions.

Venn diagrams have a single terminal relationship, that of containment. The terminal relationship is represented formally with the following schema generic over the class CBlob. The schema consists of the terminal Venn diagram components together with a contains relation between the blobs.

\[
\text{TVennDiagram}[\text{CBlob}]
\]

\[
\begin{align*}
\text{TVennDiagramComponents}[\text{CBlob}] \\
\text{contains} : \text{blobs} \leftrightarrow \text{blobs} \\
\end{align*}
\]

\[
\forall b_1, b_2, b_3 : \text{blobs} \\
\quad \neg b_1 \text{ contains } b_1 \\
\quad \land \quad (b_1 \text{ contains } b_2 \land b_2 \text{ contains } b_3 \Rightarrow b_1 \text{ contains } b_3)
\]

The condition states that the contains relation is:

- Irreflexive
- Transitive
Non-terminals are added to Venn diagrams by tagging sets of blobs. Formally, the addition of non-terminals can be expressed using the schema below, generic over the classes CBlob and CTag. It consists of the schema TVennDiagram, together with the component tag, which is a function from the sets of blobs to the class CTag.

\[
\text{TVennDiagram}\ [\text{CBlob}] \\
\text{tag} : \mathcal{P}\text{blobs} \rightarrow \text{CTag}
\]

\[
\forall B_1, B_2 : \text{dom}(\text{tag}) \\
B_1 \cap B_2 = \emptyset
\]

The condition states that no two sets of tagged blobs intersect.
An example of a non-terminal Venn diagram is presented in the figure below. Patterns are used as tags to indicate non-terminals.

Following Section 9.2 this diagram can be represented in table form as follows:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>blobs</td>
<td>{A,B,C,D,E,F}</td>
</tr>
<tr>
<td>contains</td>
<td>{(A,C),(A,D),(B,D),(B,E),(B,F)}</td>
</tr>
<tr>
<td>tag</td>
<td>{(C), (E,F)}</td>
</tr>
</tbody>
</table>
A Venn grammar can now be presented by outlining a start symbol and some production rules. The following is an example of a Venn grammar consisting of a start symbol and two production rules.
The following is a simple graphical representation of a derivation:

A Venn diagram consisting of three enclosed blobs can be generated from the above grammar.

The sequence of derivations is presented in Figure 9.14.

In conclusion, this section has illustrated that the generalised notion of grammatical formalism can be employed to develop novel grammatical formalisms and presents further evidence of the consistency and completeness of the generalised notion. However, the Venn grammars example presented here is very simple and of no immediate practical application. Chapter 10 employs the generalised notion of grammatical formalism to specify the syntax of a currently employed language and, thus provides stronger evidence for the completeness of the generalised notion.
Figure 9.14 The Generation of a Venn Diagram.
9.7 Conclusion

The conclusion to this chapter summarises the chapter and its achievements and assesses these achievements using the criteria established in the introduction (Chapter 1).

9.7.1 Summary

This chapter has developed the aspects of the formal framework concerned with specifying the syntax of languages. The chapter has used Z and the categorial extensions developed in Chapter 5.

A broad range of existing grammatical formalisms were considered in Section 9.1. The syntactic structure classes that formed the basis of these formalisms were defined using Z in Section 9.2. In Section 9.3, a generalised notion of syntactic structure class was defined, and evidence for the consistency and completeness of the generalised notion was presented using the syntactic structure classes of Section 9.2.

The concept of terminal and non-terminal categories, that form the basis for the generalised definitions of grammar and derivation were developed in Section 9.4. In Section 9.5 the generalised notions of grammar and derivation were defined, and evidence for the consistency and completeness of the generalised notions was presented by showing representations of the derivations of Section 9.1. Finally, further evidence for the consistency and completeness of the generalised notion was presented in Section 9.6, in which the generalised notion was employed to develop a new grammatical formalism.

9.7.2 Assessment

Recall that Chapter 1 outlined the following criteria for assessing the thesis that are applicable to the framework aspects of the thesis:

- Internal consistency between conception and framework
- External synthesis at the level of the framework
- Knowledge incremented at the level of the framework
This chapter takes a more detailed look at one of the aspects of language worksystem specification (Chapter 8), that of language syntax definition. As such, the consistency between the aspects of the formal framework described in this chapter and the conception, is a product of the consistency between Chapter 8 and the conception.

This chapter has sampled widely from the available literature on syntax and grammars and synthesises the aspects of this work in order to develop the generalised notion of grammar. Thus, the chapter demonstrates a high degree of external synthesis.

The generalised notion also marks a significant increment of knowledge at the level of the framework. It provides a basis for future work in defining grammatical formalisms and thus specifying the syntax of formal languages.

Further work on the framework might concentrate on two aspects. Firstly, there is scope for providing better tools for specifying individual syntactic structure classes than the Z mechanism employed in the chapter. Such a mechanism might enable easier combination of syntactic structure classes. Secondly, aspects of parsing and generation, by both human and other agents, needs to be explored. Results could be established, based upon the generalised notion and properties of syntactic structure classes, concerning issues such as the computational complexity of the parsing algorithms. Such results would then be applicable to grammatical formalisms defined using the generalised notion.
10 The Syntax of QOC

This chapter uses the generalised notion of grammatical formalism developed in Chapter 9 to extend the notion of graph grammars. The extended notion is then employed to define the syntax of QOC design rationale [MacLean et al 91]. Thus, this chapter provides further evidence of the completeness of the generalised notion of grammatical formalism. It contains the following subsections:

- Introducing Questions, Options and Criteria
- Labelled Graph Grammars
- The Syntax of QOC
- Formally Deriving the Example QOC

10.1 Introducing Design Rationale

Questions, Options and Criteria (QOC) is a graphical means for representing an analysis of a design space. Figure 10.1 is an example of QOC applied to the design problem of choosing a hardware platform:

![QOC Diagram]

A QOC diagram begins with one or more questions. In this case, the question is 'What hardware platform?'. For each question a number of different options are considered. In this case, there are two: PC and Mac. The options are assessed according to different criteria. The two options here are assessed according to the criteria of User Costs and Device Costs. The lines connecting
the options to the criteria indicate the assessment of the options in terms of the criteria. A positive assessment is represented with a solid line and a negative assessment with a dotted line. Thus, the above diagram asserts that the Mac option is assessed positively in terms of the user cost criteria but negatively against the computer cost criteria.

Criteria can be related to more general criteria. An example of this relationship can be seen in figure 10.1. In this figure, both the low user cost and low device cost are bridged to the low total cost criteria.

Options can give rise to questions. Thus, in the above diagram the PC option gives rise to a question of what central processor unit (CPU) to use. The options given are the 386 processor and the 486 processor. These options are, in turn, assessed in terms of low cost and speed.

10.2 Labelled Graph Grammars

QOC diagrams are a special case of object and arrow labelled graphs. Thus, to define their syntax, it is necessary to define a syntactic structure class for such graphs.

The basic components of such a graph can be described with the following schema generic over the class CElements. It contains a single component, elements which is a sub-set of the class CElements:

\[
\text{LGraphComponents}[\text{CElements}]
\]

\[
\text{elements} : \mathcal{P} \text{CElements}
\]

It has no conditions.

The terminal relationships of the graph can be described with the following schema generic over the classes CElements and CLabel. It consists of the LGraphComponents schema together with six additional components. The objects and arrows components are sub-sets of the elements set. The source and target components are functions from the arrows set to the objects set.
The oLabel component is a function from the objects to the class CLabel. The aLabel component is a function from the arrows to the class CLabel.

\[
\begin{align*}
\text{TLGraph}[\text{CElements, CLabel}] & \text{______________________________} \\
\text{LGraphComponents}[\text{CElements}] & \\
\text{objects} : & \mathcal{P}\text{elements} \\
\text{arrows} : & \mathcal{P}\text{elements} \\
\text{source} : & \text{arrows} \rightarrow \text{objects} \\
\text{target} : & \text{arrows} \rightarrow \text{objects} \\
\text{oLabel} : & \text{objects} \leftrightarrow \text{CLabel} \\
\text{aLabel} : & \text{arrows} \leftrightarrow \text{CLabel} \\
\end{align*}
\]

\[
\begin{align*}
\text{objects} \cup \text{arrows} &= \text{components} \\
\text{objects} \cap \text{arrows} &= \emptyset
\end{align*}
\]

The conditions state that the objects and arrows sets form a partition of the elements set.

The non-terminal relationships can be described with a schema generic over the classes CElements, CLabel and CTag. The schema consists of the TLGraph schema, together with two additional components. The oTag component is a function from objects to the class CTag. The aTag component is a function from arrows to the class CTag.

\[
\begin{align*}
\text{NTLGraph}[\text{CElements, CLabel, CTag}] & \text{___________________________} \\
\text{TLGraph}[\text{CElements, CLabel}] & \\
\text{oTag} : & \text{objects} \leftrightarrow \text{CTag} \\
\text{aTag} : & \text{arrows} \leftrightarrow \text{CTag} \\
\end{align*}
\]

\[
\begin{align*}
\text{dom(oLabel)} \cup \text{dom(oTag)} &= \text{objects} \\
\text{dom(oLabel)} \cap \text{dom(oTag)} &= \emptyset \\
\text{dom(aLabel)} \cup \text{dom(aTag)} &= \text{arrows} \\
\text{dom(aLabel)} \cap \text{dom(aTag)} &= \emptyset
\end{align*}
\]
The conditions state that:

- The domains of the \( o\text{Label} \) and \( o\text{Tag} \) functions partitions the objects set.
- The domains of the \( a\text{Label} \) and \( a\text{Tag} \) functions partitions the arrows set.

As an example of an object and arrow labelled graphs with non-terminal tags, consider figure 10.2. The objects are labelled with text and the arrows with line styles. The non terminal tags are represented with the attached grey ovals.

![Figure 10.2 An Object and Arrow Labelled Graph with Non-terminal Tags](image)

This representation can be re-expressed in tabular form as follows:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>elements</td>
<td>{e1,e2,e3,e4,e5,e6,e7,e8}</td>
</tr>
<tr>
<td>objects</td>
<td>{e1,e2,e3,e4}</td>
</tr>
<tr>
<td>arrows</td>
<td>{e5,e6,e7,e8}</td>
</tr>
<tr>
<td>source</td>
<td>{(e5,e1),(e6,e1)(e7,e2),(e8,e3)}</td>
</tr>
<tr>
<td>target</td>
<td>{(e5,e2),(e6,e3)(e7,e4),(e8,e4)}</td>
</tr>
<tr>
<td>oLabel</td>
<td>{(e1,'Q: What hardware platform'),((e2,'O: Mac'))}</td>
</tr>
<tr>
<td>aLabel</td>
<td>{(e5,—),(e6,—)}</td>
</tr>
<tr>
<td>oTag</td>
<td>{(e3,Option),(e4,Criteria Set)}</td>
</tr>
<tr>
<td>aTag</td>
<td>{(e5,Assessment),(e6,Assessment)}</td>
</tr>
</tbody>
</table>
10.3 The syntax of QOC

This section lists the start symbol and the production rules for a grammar that defines the syntax of the language of QOC. The start symbol is the following non-terminal labelled graph:

```
QOC
```

The production rules and their names are listed according to the non-terminal of the left-hand side as follows:

```
QOC
Question
Option
Assessment
Criteria Set
Criterion
Text
```

Production rules with the symbol <Text> are in fact production rule schemas. That is, they define a set of production rules. Since <text> stands for any piece of text.

QOC 1
QOC 2

Question 1

Question

Q: <text>

Option 1

Option

O: <text>

Option 2

Option

QOC
10.4 Formally Deriving the Example QOC

The production rules of the previous section are now used to derive the example presented in the introduction.

The derivation begins with the start symbol:

Rule QOC 1 is applied:
Rule Question 1 is applied:

Rule Option 1 is applied:

Rule Option 2 is applied:
Rule Option 1 is applied:

Rule Criteria Set 2 is applied:

Rule Criteria Set 1 is applied:
Rule Criterion 1 is applied:

Rule Criteria Set 3 is applied:
Rule Criteria Set 1 is applied:

Q: What hardware platform?

O: Mac

O: PC
Rule Criterion 1 is applied:

Rules Criteria Set 1 and Criterion 1 are applied in the same fashion:
Rules Assessment 1 and Assessment 2 are each applied twice:

Rule QOC 1 is applied:
Finally, the production rules are applied in much the same fashion to achieve the final result.

10.5 Conclusion

This chapter has illustrated that the generalised notion of a grammar can be used to construct a grammatical formalism, which can, in turn, be used to develop a specification of the syntax of an existing graphical language - QOC. Thus, it provides further evidence of the completeness of the general definition of grammatical formalisms.

Chapters 6 and 7 have considered the specification of performance requirements for language worksystems. Chapter 8 has considered Language Worksystems as a whole. Chapters 9 and 10 have considered the specific topic of the specification of the syntax of languages. Chapter 11 considers the form of knowledge that should support formal language design.
11 Engineering Principles

This chapter develops the aspects of the formal framework concerned with the knowledge, in the form of principles, that can be used to support language design. It is an instantiation of the Conception of Formal Language Engineering developed in Chapter 4. It is based upon the basic concepts of category theory introduced in Chapter 5.

According to Long & Dowell [Long & Dowell 89], the knowledge, in the form of principles, of an engineering discipline is defined, operationalised, tested and generalised with respect to the discipline design problem. The orientation of the knowledge of an engineering discipline with respect to the general design problem, leads to the development of principles with guaranteed effectiveness. This chapter employs category theory to develop the aspects of the framework concerned with principles.

The first section considers the nature of principles in the light of the conception of formal language engineering developed in Chapter 4, and the formal framework, as instantiated so far in Chapters 5 to 10. The second section considers how aspects of the framework can be modelled by category theory and how the notions of principle definition and operationalisation can be considered in these contexts. The third section considers how generalisable principles can be developed. The fourth section considers how the generalised knowledge embodied in engineering principles can be tested with respect to the problem of design.
11.1 The Nature of Principles

This section considers the nature of principles in the light of the conception of formal language engineering developed in Chapter 4 and the formal framework as instantiated in Chapters 5 to 10.

The first sub-section considers the class-instance relationships that exist between the different components of the framework for formal language design. The second sub-section considers how the different relationships between components of the framework give rise to the different components of principles. The third sub-section considers the distinction, arising from the work of Dowell and Long [Dowell & Long 89], between substantive and methodological principles. The fourth sub-section considers the relationships between principles.

11.1.1 Class-Instance Relationships

This sub-section considers the class-instance relationships that exist between the different components of the framework for formal language design. The formal relationships are considered first, followed by the empirical relationships.

Figure 11.1 shows the formal aspects of the generic engineering framework. It consists of four specification components and eight derivation and verification relationships between components. Each of the relationships is of a class-instance type. The relationships are discussed, in turn, below. Requirements specifications are considered first, then artifact specifications and, finally, the relationship between requirements specifications and artifact specifications is considered.

Of the eight relationships of Figure 11.1, four are formal derivation relationships and four verification relationships. If a relationship is a formal derivation, then there is an explicit technique for developing the component at the end of the arrow from the component at the beginning. If a relationship is a formal verification, then there is an explicit technique for determining if the component at the beginning of the arrow satisfies the appropriate class-instance relationship with the component at the end of the arrow.
Chapter 6 developed the aspects of the formal framework concerned with specifying the performance requirements of language worksystems. The performance of a language worksystem forms a Specific Requirements Specification of formal language engineering. Since any given general specification may be the generalisation of specific specifications, there is a class-instance relationship between General Requirements Specifications and Specific Requirements Specifications. Thus, the General Requirements Specification is a class of performance requirements.

An illustration can be provided by planning and control. One might construct performance requirements for generic time-based planning and control such that the performance requirements for air traffic control were an instance of the generic performance requirements.

General Requirements Specifications may be considered of as a generalisation of Specific Requirements Specifications, consisting of generic sets of desired performances which are, in turn, combinations of generic desired benefits and desired costs. Generic Requirements Specifications will also consist of generic performance orderings which are, in turn, combinations of generic benefit and cost orderings.

Chapter 9 was concerned with developing the aspects of the framework concerned with specifying formal language worksystems. A formal language worksystem forms a Specific Artifact Specification of formal language engineering. Since any given general specification may be the generalisation of specific specifications, there is a class-instance relationship between
General Artifact Specifications and Specific Artifact Specifications. Thus, the General Artifact Specification is a class of formal language worksystems.

An illustration can be provided by planning and control. One might construct a generic time-based planning and control language worksystem such that a particular air traffic control system was an instance of the generic system.

General Artifact Specifications may be considered as a generification of Specific Artifact Specifications, consisting of generic formal languages and generic language agents. Since the specific language agents are of the same form as Specific Requirements Specifications, that is desired performances, generic language agents are of the same form as General Requirements Specifications.

From above, General Requirements Specifications are of the form of generic performance requirements, and General Artifact Specifications are of the form of generic formal language worksystems. Postulating an engineering principle of formal language design involves establishing a relationship between some generic performance requirements specification and some generic formal language worksystem specification. For any given generic performance requirements specification, there may be many formal language worksystem specifications that satisfy those performance requirements. Thus, there is a class-instance relationship between General Requirements Specifications and General Artifact Specifications.

An illustration can be provided by planning and control. One might construct relationships between a generic performance requirements specification for time-based planning and control and generic time-based planning and control language worksystem specifications, such that the generic language worksystem specification satisfied the generic performance requirements specification.

The relationships between General Requirements Specifications and General Artifact Specifications indicate how generic formal language specifications can be used to combine generic language agent specifications and the generic actual performance that results from such combinations. Some indications of how such combinations might be represented using co-cones of category theory was presented in Section 8.3.4.
Specific Requirements Specifications are of the form of performance requirement specifications and Specific Artifact Specifications are of the form of formal language worksystem specifications. Applying an engineering principle of formal language design involves establishing a relationship, between some performance requirement specification and some formal language worksystem system specifications, that is based upon some relationship between some generic performance requirements specification and some generic formal language worksystem specification. For any given performance requirements specification, there may be many formal language worksystem specifications that satisfy those performance requirements. Thus, there is a class-instance relationship between Specific Requirements Specifications and Specific Artifact Specifications.

An illustration can be provided by planning and control. Given: a relationship between generic performance requirements for time-based planning and control; a generic time-based planning and control language worksystem such that the generic language worksystem satisfied the generic performance requirements; a performance requirements specification for air traffic control that is an instance of the generic performance requirements; and an air traffic control system that was an instance of the generic formal language worksystem, it is possible to develop a relationship between the performance requirements for air traffic control and the air traffic control worksystem.

As for the general case, the relationships between Specific Requirements Specifications and Specific Artifact Specifications indicate how specific formal languages can be used to combine specific language agents and the specific actual performance that results from such combinations.

From Figure 11.1, there may be two ways in which a Requirements Specification is related to Artifact Specifications. If the relationship is a formal derivation from the General Requirements Specification to the General Artifact Specifications, then there is an explicit technique for developing Specific Artifact Specifications which are instances of the Specific Requirements Specification, provided it is demonstrated that the Specific Specifications are instances of the General Specifications. If the relationship is a formal verification, then there is an explicit technique for determining if a Specific Artifact Specification is an instance of a Specific Requirements
Specification, with the same proviso relating Specific and General Specifications.

The distinction between formal derivation and verification relationships is key in distinguishing between substantive and methodological principles. The distinction is outlined in Section 11.1.3 below.

The discussion of principles has concentrated upon the formal aspects of principles. However, principles should also have an empirical aspect. Figure 11.2 shows the empirical aspects of the generic engineering framework. It consists of two specification components, the specific specifications, two empirical components, the Client Requirements and Artifact and six derivation and validation relationships between components. Each of the relationships is of a class-instance type. The relationships are discussed, in turn, below. The requirements components are considered first, then the artifact components. Finally, the relationship between the Client Requirements and Artifact is considered.

Of the six relationships of Figure 11.2, three are empirical derivation relationships and three empirical validation relationships. If a relationship is an empirical derivation, then there is an explicit technique for developing the component at the end of the arrow from the component at the beginning. If a relationship is an empirical validation, then there is an explicit technique for determining if the component at the beginning of the arrow satisfies the appropriate class-instance relationship with the component at the end of the arrow.

![Figure 11.2 The Empirical Aspects of the Generic Engineering Framework](image-url)
Chapter 6 developed the aspects of the formal framework concerned with specifying the performance requirements of language worksystems. The performance requirements of a language worksystem forms a Specific Requirements Specification of formal language engineering. Since any performance requirements specification is an abstraction of the Client Requirements, there is a class-instance relationship between Specific Requirements Specifications and Client Requirements. Thus, the Specific Requirements Specification represents a class of Client Requirements. The performance requirements consist of the set of desired performances and the performance ordering which relates performances. The set of desired performances represents the set of possible performances desired by the client. The performance ordering enables performances to be related in such a way as to take account of the potentially conflicting requirements of the client.

An illustration can be provided by air traffic control. One might specify performance requirements for air traffic control, where the clients was, for example, a civil aviation authority. The client requirements would consist of the conflicting goals of the authority, those of safety and expediency, and in addition other requirements such as the health and safety aspects concerning human controllers. The performance requirements developed for one aviation authority may be applicable to another authority, thus giving rise to the class instance relationship.

Chapter 8 developed the aspects of the formal framework concerned with specifying language worksystems. The language worksystem specification forms a Specific Artifact Specification of formal language engineering. Since any language specification is an abstraction of the Artifact, there is a class-instance relationship between Specific Artifact Specifications and Artifact. Thus, the Specific Artifact Specification represents a class of Artifacts. A formal language worksystem specification consists of the set of formal language specifications and a set of language agent specifications. The language agent specifications represent the components of the Artifact. The language specifications are necessary in order to specify how the components of the Artifact communicate.
An illustration can be provided by air traffic control. One might specify a formal language worksystem for air traffic control consisting of two language agents (one human and one a computer) together with a language by which the agents communicate. The Artifact would consist of a human and a computer both carrying out transformations of work and employing and interpreting the formal language according to the specification. There may be many such humans and computers that could form the Artifact, thus giving rise to the class-instance distinction.

The final relationships considered are those between the Artifact and the Client Requirements. Since there may be many Artifacts which satisfy Client Requirements, there is a class-instance relationship between Client Requirements and Artifact. Thus, the Client Requirements can be considered as giving rise to a class of Artifacts.

An illustration can be provided by air traffic control. An actual air traffic control worksystem consisting of a human and a computer, both carrying out transformations of work and employing and interpreting some formal language, might satisfy the Client Requirements of a civil aviation authority. Clearly there may be many such worksystems that could satisfy the Client Requirements, thus giving rise to the class-instance distinction.

This empirical validation relationship between Artifact and Client Requirements is central to the development of principles, forming the means by which principles can be tested with respect to the problem of design. The issue of principle validation is discussed further in Section 11.4.

This sub-section has considered the nature of the class-instance relationships that exist between the different components of the framework for formal language design. The next section considers how these different relationships give rise to the different components of a principle.

**11.1.2 Components of a Principle**

This sub-section considers how the different relationships between components of the framework give rise to the different components of a principle.
Principles consist of general knowledge that can be applied to solve specific problems that arise with the requirements of clients. Principles consist of formal and empirical components. The formal components of a principle arise in relating both general and specific specifications. The empirical components arise in relating specific specifications to the requirements of the client and the artifact that fulfils that specification. The components are defined below and exemplified in terms of a hypothetical planning and control principle.

The formal relationship between a General Requirements Specification and a General Artifact Specification is an essential component of a principle. This relationship may be called the formal component of the principle. The formal component of the principle is embedded within the formal engineering framework. The framework enables the general specifications to be related to specific specifications, and thus, the establishing of a relationship between Specific Requirements Specification (the performance requirements specifications for language worksystems) and Specific Artifact Specification, (language worksystem specifications)

The formal component of a time-based planning and control principle would consist of: a generic performance requirements specification for time-based planning and control; one or more generic language worksystems for time-based planning and control; and relationships between the generic language worksystems and the generic performance requirements indicating the performance of the worksystems. The formal component would be embedded in the framework so as to enable the establishing of relationships between the generic performance requirements and specific performance requirements such as those for an air traffic control system. The framework would also enable relationships to be established between the generic language worksystems and specific language worksystems.

Any principle must also determine the acceptable empirical derivation and validation techniques that must be employed to establish a relationship between Client Requirements and Specific Artifact Specifications. This set of techniques may be called the requirements capture component of the principle.

The requirements capture component of a time-based planning and control principle would specified the methods that were acceptable in relating
performance requirements specifications to the requirements of clients. Thus, in order to apply the general principle to the specific case of air traffic control, the performance requirements specification for the air traffic control system would have to be related to the Client Requirements using a method specified in the requirements capture component of the principle.

Analogously, a principle must describe the acceptable empirical derivation and validation techniques that must be employed to establish a relationship between Specific Artifact Specification and Artifact. This set of techniques may be called the implementation component of the principle.

The implementation component of a time-based planning and control principle would specify the methods that were acceptable in relating language worksystem specifications to actual language worksystems. Thus, in order to apply the general principle to the specific case of air traffic control the language worksystem specification for the air traffic control system would have to be related to the actual language worksystem using a method specified in the implementation component of the principle.

Finally, the principle must specify the empirical derivation and validation techniques that must be employed to determine whether an Artifact that results from the application of an engineering principle satisfies the Client Requirements to which the principle was applied. This set of techniques may be called the validation component of the principle. The validation component of the principle imposes the conditions under which the principle provides its guarantee of effectiveness.

The validation component of a time-based planning and control principle would specify the methods that were acceptable in relating actual language worksystems to the requirements of clients. The generic planning and control principle attempts to guarantee that, if the formal relationships are established according to the formal component of the principle and the formal framework, and the relationship between the air traffic control performance requirements and the requirements of the client are established according to the requirements capture component, and the relationship between the air traffic control worksystem specification and the actual air traffic control worksystem are established according to the implementation component,
then all the techniques specified in the validation component of the principle will establish a relationship between Artifact and Client Requirements.

This sub-section shows how the different relationships between components of the framework give rise to the different components of principles. A principle was said to consist of a formal component embedded within the formal framework, together with the three empirical components of requirements capture, implementation and validation. The next section considers the distinction between substantive and methodological principles.

11.1.3 Substantive and Methodological Principles

This sub-section considers the distinction, arising from the work of Dowell and Long [Dowell & Long 89], between substantive and methodological principles. Essentially, a methodological principle details how to solve a particular design problem. A substantive principle does not state how a problem should be solved, but rather, the properties of an artifact that is a solution to the problem.

Given the components of an engineering principle outlined above. A principle can be said to be methodological, if the following hold:

- The formal component of the principle is a derivation.
- The requirements capture component specifies an empirical derivation method.
- The implementation component specifies an empirical derivation method.

The empirical derivation method of the requirements capture component must either produce a performance requirements specification, which is an instance of the generic performance requirements specification of the principle, or fail to produce any performance requirements specification at all. If no performance requirements specification is produced, then the principle does not apply.
The empirical derivation method of the implementation component must produce an artifact, for all possible language worksystems that are instances of the generic language worksystem of the principle.

If the hypothetical planning and control principle considered above was methodological, then once the performance requirements for the air traffic control system had been established, it follows, according to the guarantee of the principle, that an actual language worksystem can be constructed which can be shown to satisfy the Client Requirements using each method in the validation component of the principle.

In contrast, for a substantive principle, since it relies upon formal verification and empirical validation techniques, it does not follow that once a performance requirements specification has been constructed that there is necessarily a language worksystem satisfies Client Requirements. Indeed if, for example, the implementation component only specifies empirical validation techniques, even if a specific language worksystem has been specified, it still does not follow that there is a language worksystem that satisfies this specification.

While methodological principles may be considered to be the most desirable, at least, in terms of the certainty they afford to the formal language designer, substantive principles, once fully applied, provide the same guarantee of performance. Indeed, it may be likely that substantive principles are able to provide guarantees of more effective performance, precisely because the existence of the performance requirements does not guarantee the existence of a language worksystem that satisfies Client Requirements. Thus, substantive principles may trade the certainty of solution for possible solutions with greater performance.

11.1.4 Relationships between Principles

Since language worksystems are composed of agents which may be developed by other disciplines, it follows that the principles of formal language engineering might be dependant upon principles of other disciplines. Furthermore, in complex problems, it may be the case that more than one principle of formal language engineering might be applicable to the
problem. This sub-section considers the relationships between principles, considering the component discipline case first.

Consider, for example, a principle of formal language engineering for which some of the language worksystems of the general artifact specification have both human and computer language agents. It should be expected that the formal language engineering principle would be based upon principles of human factors and software engineering that guarantee the performance of the individual language agents.

In this case, once the specification of a specific language worksystem has been established, there will be specifications of both human and computer agents. These specifications form the requirements of the client for the disciplines of human factors and software engineering respectively. The specification of language agents takes the same form as requirements specifications of the worksystem as a whole, to take account of the situation where one discipline's artifact specification forms the requirements specification of another discipline.

In such a situation, the principles of the component disciplines form the relationships between the language worksystem specification and the actual language worksystem. In other words, between the Specific Artifact Specification and the Artifact. Principles of component disciplines thus form some of the empirical techniques that make up the implementation component of the formal language engineering principle. Methodological principles of component disciplines give rise to empirical derivations, whereas substantive principles give rise to empirical validations.

Consider, for example, the hypothetical planning and control principle outlined above. This principle may specify a generic computer language agent as part of the generic formal language worksystem. The discipline of software engineering may provide a substantive principle concerning such generic computer language agents. The substantive principle of software engineering would form an empirical validation technique of the planning and control principle.

In addition to the principles of component disciplines, there is a need to be able to combine principles within formal language engineering itself. Such a
situation would arise when more than one principle applies to a given Client Requirements.

Consider two principles. First consider a generic planning and control principle as outlined above. Second consider a principle concerned with the manipulation of objects in three dimensional space. Both of these principles might apply to some aviation authority's requirements for an air traffic control system.

The formal framework should therefore provide a means by which principles can be applied in combination in such situations. In simple cases, combined application of principles may be straightforward. For example, if a language worksystem consisted of two agents which were in themselves language worksystems, such that there was no communication between the two agents, different principles might be freely applied to the design of each agent. However, it is important to note that, in general, in combined application of principles, there are likely to be some dependencies between the components of the principle. Such dependencies must be taken into account, in predicting the performance guarantees that are the result of combined principle application.

This sub-section has considered the relationships between principles. For the case of component disciplines, it was claimed that the principles of component disciplines correspond to the techniques of the implementation component of the formal language engineering principle. For the case of relationships between principles of formal language engineering, it was claimed that the formal framework provide a means for the combined application of principles.

11.1.5 Conclusion

To conclude the discussion of principles in this section, the first sub-section considered the formal and empirical aspects of principles through a discussion of the class-instance relationships that arise in the formal language engineering framework. The second sub-section employed the work of the previous section to introduce the four components of a principle. These components are the formal, requirements capture, implementation and
validation components. The third sub-section considered the nature of substantive and methodological principles. The fourth sub-section considered the relationships between principles of formal language engineering, as well as the relationships between principles of other disciplines and those of formal language engineering.

In what follows, the second section considers how aspects of the framework can be modelled by category theory and how the notions of principle definition and operationalisation can be considered in these contexts. This section builds upon the discussion of class-instance relationship. The third section considers how generalisable principles can be developed, considering not only how principles can be developed from craft and applied science activity, but, also, how principles can be formed by combining existing principles. The fourth section considers how the generalised knowledge embodied in engineering principles can be tested with respect to the problem of design. Key in this consideration will be the relationship between the four components of a principle.
11.2 A Categorial Meta View of Engineering Design

This section considers how aspects of the framework can be modelled by category theory and how the notions of principle definition and operationalisation can be considered in these contexts. It is based largely upon the discussion of class-instance distinctions from Sub-section 11.1.1.

The first sub-section models the class-instance distinction between General and Specific Requirements Specifications. The second sub-section models the class instance distinction between General and Specific Artifact Specifications. The third sub-section combines the models developed in the two previous sub-sections to present a model of all the formal class-instance relationships of the framework. The fourth sub-section considers the formal aspects of principle definition within the model established in the third sub-section. The fifth sub-section considers the formal aspects of principle operationalisation within the model established in the third sub-section.

11.2.1 The Category of Requirements Specifications

This sub-section models the class-instance distinction between General and Specific Requirements Specifications.

In the formal language engineering framework, both Specific and General Requirements Specifications are represented in the form of performance requirements over some work structures and worksystem costs (Section 6.3). The set of all such performance requirements may be considered as the set of objects of the Category of Requirements Specifications.

The arrows of the Category of Requirements Specifications represent the class-instance relationships between performance requirements. Thus, if \( p \) and \( p_1 \) are performance requirements specifications, then there is an arrow from \( p \) to \( p_1 \) if, and only if, \( p \) is an instance of \( p_1 \).

Since the Category of Requirements Specification is a category, there are identity arrows between each object and an operation for composing arrows. The identity arrows indicate that each performance requirements specification is considered to be an instance of itself. The composition of arrows indicate that if \( p, p_1 \) and \( p_2 \) are performance requirements such that, \( p \)
is an instance of $p_1$, and $p_1$ is an instance of $p_2$, then $p$ is an instance of $p_2$. These assumptions concerning identities and composition are justified, since all performance requirements specifications, both generic and specific are of the same form.

Figure 11.3 represents a fragment of the Category of Requirements Specifications, in which identity arrows and composite arrows are not represented for reasons of clarity. The objects $p$ and $p'$ represent specific performance requirements specifications. Any performance requirements specification $p$ is specific if, and only if, for any object $p'$ and arrow $r$, if $r$ is an arrow from $p'$ to $p$, then $p' = p$ and $r = id(p)$. The Category of Requirements Specifications enables performance requirements specifications to be considered as a class-instance hierarchy. Thus, $p_2$ is a generalisation of $p_1$, which is, in turn, a generalisation of $p$.

![Figure 11.3 A Fragment of the Category of Requirements Specifications](image)

The definition of the Category of Requirements Specifications that has been presented here is informal, since although the class of the objects has been formally defined in Section 6.3, the class of arrows has only been defined informally. For the purposes of this chapter, considering the nature of principles, this informality will not be important. The implications for the framework as a whole are discussed in the conclusion of the chapter.
11.2.2 The Category of Artifact Specifications

This sub-section models the class-instance distinction between General and Specific Artifact Specifications.

In the formal language engineering framework, both Specific and General Artifact Specifications are represented in the form of language worksystems, over some work structures and syntactic structure classes (Chapter 8). The set of all such language worksystems may be considered as the set of objects of the Category of Artifact Specifications.

The arrows of the Category of Artifact Specifications represent the class-instance relationships between language worksystems. Thus, if \( l \) and \( l_1 \) are language worksystem specifications, then there is an arrow from \( l \) to \( l_1 \) if, and only if, \( l \) is an instance of \( l_1 \).

As for the Category of Requirements Specifications, identity arrows indicate that each Performance Requirements Specification is considered to be an instance of itself, and composition of arrows indicate that if \( l \), \( l_1 \) and \( l_2 \) are performance requirements, such that \( l \) is an instance of \( l_1 \) and \( l_1 \) is an instance of \( l_2 \), then \( l \) is an instance of \( l_2 \). Again these assumptions concerning identities and composition are justified, since all language worksystem specifications, both generic and specific, are of the same form.

Figure 11.4 A Fragment of the Category of Artifact Specifications

Figure 11.4 represents a fragment of the Category of Artifact Specifications, in which identity arrows and composite arrows are not represented for reasons of clarity. The objects \( l \) and \( l' \) represent specific language worksystem
specifications. As for the Category of Requirements Specifications, any language worksystem specification \( l \) is specific if, and only if, for any object \( l' \) and arrow \( r \), if \( r \) is an arrow from \( l' \) to \( l \), then \( l' = l \) and \( r = \text{id}(l) \). The Category of Artifact Specifications enables language worksystem specifications to be considered as a class-instance hierarchy. Thus, \( l_2 \) is a generalisation of \( l_1 \) which is, in turn, a generalisation of \( l \).

The definition of the Category of Artifact Specifications that has been presented here is informal. Indeed in a sense it is even more informal than the Category of Requirements Specifications since not even the class of the objects, has been formally defined in the thesis. For the purposes of this chapter, considering the nature of principles, this informality will not be important. The implications for the framework as a whole are discussed in the conclusion of the chapter.

**11.2.3 The Framework Category**

This sub-section combines the models developed in Sub-sections 11.2.2 and 11.2.3 to present a model of all the formal class-instance relationships of the framework.

In the formal language engineering framework, class-instance relationships are postulated between artifact and requirements, at both the general and specific levels. The Framework Category models these relationships together with relationships between requirements and artifact specifications.

The objects of the Framework Category are the objects of the Category of Requirements Specifications together with the objects of the Category of Artifact Specifications.

The arrows of the Framework Category consist of the arrows of the Category of Requirements Specifications, the arrows of the Category of Artifact Specifications, together with the arrows that represent the relationships between artifacts and requirements. Thus, if \( l \) is a language worksystem specification and \( p \) is a performance requirements specification, then there is an arrow from \( l \) to \( p \) if, and only if, \( l \) satisfies \( p \).
Since all the objects of the Framework category come from either the Category of Requirements Specifications or the Category of Artifact Specifications, and the Framework Category contains all the arrows of the other two categories, the existence of identity arrows is guaranteed. The composition of arrows to the Framework Category is extended, so that if generic language worksystem specification $l_1$ satisfies generic performance requirements specification $p_1$, and language worksystem specification $l$ is an instance of $l_1$, then $l$ also satisfies $p_1$. Figure 11.5 represents a fragment of the Framework Category, in which identity arrows and composite arrows are not represented for reasons of clarity.

Given the direction of the arrows of the Framework Category, it can be seen that only the verification aspects of the formal framework are being modelled. It is possible to extend the consideration to formal derivations. This extension is not presented here in order to simplify what follows.
11.2.4 The Formal Aspects of Principle Definition

This sub-section considers the formal aspects of principle definition within the model established in the Sub-section 11.2.3.

In the Framework Category, the arrows between requirements specifications and the arrows between artifact specifications arise due to the nature of requirements and artifact specifications. The arrows between artifact specifications represent the formal components of principles and their application. Thus, the definition of the formal component of a principle involves the addition of a new arrow to the Framework Category.

Adding an arrow to a category will almost certainly necessitate revising the composition operator to take account of the new arrow. In adding the arrow which describes the formal component of a principle, it is necessary that the relationship between generic language worksystem specifications and generic performance requirements specifications is defined in such a manner as to make it possible to determine any resulting compositions.

Figure 11.6 represents a fragment of the Framework Category, in which identity arrows and composite arrows are not represented for reasons of clarity. In defining the formal component of a principle, it is necessary to specify a relationship r1 from worksystem l1 to requirements p1 such that, the composition of r1 with any relationships such as r and r' is defined.

![Figure 11.6 Defining the Formal Components of Principles extends the Composition Operator in the Framework Category](image_url)
11.2.5 The Formal Aspects of Principle Operationalisation

This sub-section considers the formal aspects of principle operationalisation within the model established in the Sub-section 11.2.3. Operationalisation is the transformation of the definitions of knowledge into a form that can be used and tested.

The formal application of a principle involves determining limits or co-limits in the Framework Category. To illustrate the use of limits and co-limits in the application of principles, the following case is considered. There is a specific performance requirements specification $p$ which is related, in the framework, with arrow $r$ to a generic performance requirements specification $p_1$. There is a principle $r_1$ from generic formal language worksystem $l_1$ to $p_1$. This situation is represented diagramatically in Figure 11.7.

![Figure 11.7 A Basis for the Formal Application of a Principle.](image)

The formal application of the principle with formal component $r_1$ involves discovering a limit (in particular, a pullback) for the diagram in Figure 11.7. This pullback is represented in Figure 11.8. It is important to note that there is no guarantee that this pullback necessarily exists in the Framework Category.
In general, the formal aspects of the operationalisation of the knowledge embodied in principles involves determining the specific satisfaction relationships that are a consequence of the general principle. Consider the diagram in Figure 11.9. The object $l_1$ is a generic language worksystem specification. The object $p_1$ is a generic performance requirements specification. The arrow $r_1$ represents the formal component of a principle. The objects $p$ and $l$ are specific performance requirements and language worksystem specifications respectively. If the diagram commutes, the arrow $r$ represents a specific satisfaction relationship between $l$ and $p$.

This sub-section has considered only the formal aspects of principle operationalisation. The empirical aspects of operationalisation instantiates the Specific Requirements Specification to Client Requirements, using the Requirements Capture component of a principle, and instantiates the Specific Artifact Specifications to Artifact, using the implementation component of the principle. The empirical and formal aspects of operationalisation are considered together in Section 11.4, which is concerned with the validation of principles.
11.2.6 Conclusion

This section has shown how the formal framework for formal language engineering can be represented using concepts from category theory. Categories of Requirements Specifications and Artifact Specifications were introduced and combined to create the Framework Category. The section outlined how principles may be defined and operationalised within the framework category. The next two sections of this chapter consider how the generalised knowledge embodied in engineering principles may be generated and validated.
11.3 Generating Engineering Principles

This section considers how generalisable engineering principles can be developed. It consists of three sub-sections. The first sub-section considers how principles can be generated from formalised craft-based design. The second sub-section considers how principles might be generated from scientific and applied science knowledge. The third section considers how principles may be generated from other principles of formal language design.

11.3.1 Principles from Formalised Craft-based Design

Principles may be generated from successful formalised craft-based design activity. Consider for example the following situation. A design has been carried out and it is a success. That is, the artifact satisfies the clients requirements. A specific performance requirements specification, p, and a specific language worksystem specification, l, have been developed and they are empirically related to the clients requirements and the actual language worksystem, respectively. Further, there is a formalised rationale for the success of the design in the form of a postulated relationship between l and p, represented in the framework category with the arrow r. Finally, there is a generic language worksystem specification, l1, and a formal verification, t, demonstrating that l is an instance of l1. The formal components and relationships of this situation are represented diagrammatically in Figure 11.10.

![Figure 11.10 A Basis for Generalising Successful Craft-based Design.](image)

In order to generate the formal component of the principle, it is necessary to construct the generic performance requirements specification, p1, and
relationships between l1 and p1, and p and p1. In categorical terms, the
required components are then a co-limit (a pushout in fact) of the above
diagram of Figure 11.10. The pushout is represented in Figure 11.11.

\[ \begin{array}{c}
  p1 \\
  s \\
  p \\
  l \\
  \end{array} \quad \begin{array}{c}
  r1 \\
  t \\
  r \\
  \end{array} \]

Figure 11.11 Generalising Successful Craft-based Design.

The approach illustrated here is really a form of inductive reasoning. In the
above example, the induction was based upon a single successful design.
There is no reason why such induction might not be performed on multiple
designs, each design forming a disjoint component of the diagram whose
limit is being constructed.

The discussion of the generation of principles from successful designs has
concentrated on the formal component of a principle. It is also necessary to
consider the empirical components. Thus, the empirical components of the
principle are, in some sense a limit, of the empirical components of each of the
successful design examples. It is important to note that, in general, the limits
for generating the formal and/or empirical components of principles may not
necessarily exist.
11.3.2 Principles from Scientific Knowledge

Scientific knowledge is defined, operationalised, generalised and tested with respect to the general problem of science, that is understanding. However, scientific knowledge is not generally defined, operationalised, generalised and tested with respect to the general problem of design. Thus, in order to obtain engineering principles from existing scientific knowledge, it is necessary to consider how such knowledge might be transformed into an appropriate form for design.

The key aspect of engineering knowledge that is not possessed by scientific knowledge is the separation between requirements and artifact. Thus in order to transform scientific knowledge into a form that can be defined, operationalised, tested and generalised with respect to the general problem of design, it is necessary to identify the three components of the scientific knowledge that correspond to the General Requirements Specification, the General Artifact Specification and the relationship between the two types of General Specification. If such components can be identified then it is possible to redefine the scientific knowledge in the form of the formal component of an engineering principle that can be operationalised and tested with respect to the general problem of design.

The discussion of the generation of principles from scientific knowledge has concentrated on the formal component of a principle. It is also necessary to consider the empirical components. Thus, the empirical components of a principle generated from scientific knowledge should be based upon the empirical techniques that were used to develop the original scientific knowledge. It is important to note that, for any given sub-set of scientific knowledge, it may not be possible to generate a corresponding engineering principle.

11.3.3 Combining Principles of Formal Language Engineering

The third means of generating principles that is considered here involves the combination of existing principles. The need for the framework to enable such combination was discussed in Section 11.1.4.
Consider a situation where a language worksystem is to achieve a performance $p$, which can generalised to two performances $p_1$ and $p_1'$. In addition, there are principles $r_1$ and $r_1'$, so that $r_1$ guarantees that a worksystem $w_1$ has performance $p_1$, and $r_1'$ guarantees that worksystem $w_1'$ has performance $p_1'$. The formal components and relationships of this situation are represented diagrammatically in figure 11.12.

![Figure 11.12 A Basis for Combining Principles](image)

In order to construct a combination of the formal components of principles $r_1$ and $r_1'$, it is necessary to construct a performance specification $p_2$, a worksystem specification $w_2$, and an arrow $r_2$ from $w_2$ to $p_2$, so that the diagram of Figure 11.13 commutes in the Framework Category.

![Figure 11.13 Combining Principles](image)

By this means, principles can be derived from other more basic principles. The approach illustrated is a form of deductive reasoning. In the above example, the deduction was based upon the combination of performance requirements, worksystems and the formal principles.
In combining principles, in order to achieve a worksystem specification with maximum effectiveness, one might wish to have the least general principle that results from the combination. In the above case, this wish would amount to requiring that $p''$ was a co-limit (in fact a pushout) of the complete subdiagram involving $p$, $p_1'$ and $p_2'$, and that $w_2$ was a co-limit (in fact a coproduct) of $w_1'$ and $w_2'$.

The discussion of the combination of principles has concentrated on the formal component of a principle. It is also necessary to consider the empirical components. Thus, the empirical components of the combined principle are, in some sense, a combination of the empirical components of the principles being combined. It is important to note that, in general, combined principles may not necessarily exist.

11.3.4 Conclusion

This section has shown how the formal framework for formal language engineering can support the generation of engineering principles. Three means of generating the formal components of engineering principles have been considered. It was illustrates how engineering principles might be generated from successful craft-based design by constructing limits and co-limits in the Framework Category. It was argued that formal scientific knowledge could be redefined in the form of engineering principles by identifying the requirements and artifact aspects of the scientific knowledge. Once this redefinition had taken place, it would be possible to operationalise and test such scientific knowledge with respect to design problems. Finally, it was demonstrated how commuting diagrams, co-limits and limits may be used to construct principles, that are a combination of existing principles. The next section considers the validation of principles.
11.4 Validating Engineering Principles

This section considers how the generalised knowledge embodied in engineering principles can be tested with respect to the problem of design. It contains three sub-sections. The first sub-section considers how the Framework Category can be extended to include the empirical relationships involving Client Requirements and Artifacts. The section considers the concept of successful principle application and how successful application acts to validate principles. The third section considers the implications of unsuccessful design for the evolution of engineering principles.

11.4.1 Extending the Framework Category

Principle validation necessarily involves the consideration of Client Requirements and Artifacts. This consideration can be achieved by defining an Extended Framework Category.

The objects of the Extended Framework Category consists of the objects of the Framework Category, together with additional objects which represent Client Requirements and Artifacts.

The arrows of the Extended Framework Category consist of the arrows of the Framework Category, together with arrows that represent the relationships between Client Requirements and Specific Requirements Specifications, Artifacts and Specific Artifact Specifications and Artifacts and Clients Requirements. Thus, if cr represents a Clients Requirements, and p is a performance requirements specification, then there is an arrow from cr to p if and only p is a performance requirements specification corresponding to Client Requirements.

While the Framework Category has not been formally defined in this thesis, the Extended Framework is necessarily informal, since it contains arrows which represent empirical relationships. These relationships are necessarily empirical since, without some meta-framework, it is not necessarily the case that Client agents and Language Designer agents share a common task (see the definition of formal in Chapter 2).
11.4.2 Successful Principle Application and Principle Validation

Engineering principles may be validated through application to design. The repeatedly successful application of a principle acts to validate the principle.

Successful design occurs when the application of the techniques of the requirements capture, formal and implementation components of the principle to the requirements of the client, produces an artifact which is validated against the client requirements, using the techniques of the validation component of the principle.

Consider the following case of the application of a principle:

- The requirements of the client are the object cr.
- The performance requirements specification is the object p
- The generic performance requirements specification is the object p1
- The generic worksystem specification is the object w1
- The worksystem specification is the object w
- The artifact is the object a.
- The arrow s represent a relationship between p and cr established with the requirements capture component of the principle.
- The arrow s1 represents a relationship between p1 and p established with the formal framework.
- The arrow r1 represents a relationship between w1 and p1, i.e. the formal component of the principle.
- The arrow t1 represents a relationship between w and w1 established with the formal framework.
- The arrow r represents a relationship between w and p established with the formal framework.
- The arrow t represents a relationship between a and w established with the implementation component of the principle.
The situation is represented in the Figure 11.14.

![Diagram](image)

**Figure 11.14 The Basis for Assessing Successful Principle Application.**

The principle has been successfully applied if, for any arrow between a and cr that represents a relationship, established with the validation component of the principle, the diagram of Figure 11.15 commutes:

![Diagram](image)

**Figure 11.15 Successful principle application.**

With this criterion, the successful application of principles does not necessarily lead to satisfied clients. If the techniques of the validation component of a principle do not correspond to the client notion of a successful design, then the client will remain unsatisfied. Thus, for successful principal application to lead to a satisfied client, it is essential that the client is satisfied with the validation techniques of the principle being employed.
11.4.3 The Implications of Unsuccessful Principle Application

This sub-section considers how unsuccessful principle application can lead to the evolution of a principle. The unsuccessful application of a principle is the result of the inadequacies of one or more of the principle components or of the formal framework. The implications of these inadequacies are discussed below.

If the failure of a principle can be traced to an inadequacy of the requirements capture component, then the requirements capture component must be improved. The requirements capture component can be improved by changing or reducing the set of acceptable empirical derivation and validation techniques specified by the component.

If the failure of a principle can be traced to an inadequacy of the formal capture component, then the formal component must be improved. The formal component can be improved by changing or reducing the set of relationships between General Requirements Specifications and General Artifact Specifications specified by the component.

If the failure of a principle can be traced to an inadequacy of the implementation component, then the implementation component must be improved. The implementation component can be improved by changing or reducing the set of empirical derivation and validation techniques, specified by the component.

If the failure of a principle can be traced to an inadequacy of the validation component, then the validation component must be improved. The validation component can be improved by changing or reducing the set of the component's empirical derivation and validation techniques, that must be employed to validate artifact against client requirements. Reducing the set of validation techniques is may to lead to an increase in the situations, in which principles are successfully applied but clients are unsatisfied, that were discussed at the end of Sub-section 11.4.2.

If the components of a principle are improved by changes to, rather that reductions of, the elements of components, then previously successful
applications of the principle may not provide evidence of the validity of the modified principle.

If the failure of a principle is due to inadequacies in the formal framework then the framework must be modified. Since modifications to the framework affect all the principles embedded in the framework, they constitute very serious undertakings. For a framework to be successful, problems of this nature should only arise early in the life of the framework.

11.4.4 Conclusion

This section has considered the validation of principles. The Extended Framework Category was outlined and the notion of successful principle application outlined in terms of the extended framework. Principles may be validated by repeated instances of successful application. Unsuccessful application of principles may lead to the modifications of one or more components of the principle or even the framework underlying the principle.
11.5 Conclusion

The conclusion to this chapter summarises the chapter and its achievements and assesses these achievements by the criteria established in the introduction (Chapter 1).

11.5.1 Summary

This chapter has developed the aspects of the formal framework concerned with knowledge, in the form of engineering principles. It employed the notions of category theory developed in Chapter 5.

The nature of principles was discussed in terms of the class-instance distinction, and the formal and empirical aspects of principles were outlined. The formal aspects of principles were embedded in the Framework Category, which was then used to consider principle definition and operationalisation. The generation of principles, from craft-based design, scientific knowledge and other principles, was considered in the terms of the Framework Category. The Extended Framework Category was introduced to model the empirical aspects of principles. The concept of successful principle application, through which principles may be validated, was outlined in terms of the Extended Framework Category.

11.5.2 Assessment

Recall that Chapter 1 outlined the following criteria for assessing the thesis that are applicable to the framework aspects of the thesis:

- Internal consistency between conception and framework
- External synthesis at the level of the framework
- Knowledge incremented at the level of the framework

The concept of principles developed in this chapter, was derived directly from the conception of formal language engineering outlined in Chapter 4, and thus, in a broad sense, these aspects of the framework are consistent with the conception. However, the informal nature of the Framework Category
and associated categories indicates the preliminary nature of much of the work of this chapter, and the lack of very strong links with the aspect of the framework that are concerned with the details of performance requirements and artifact specifications. The aim of formalising principles in terms of the formal definitions of performance requirements specifications and worksystem specifications should be an early aim in an attempt to continue this work.

It can be argued that the failure to present a concrete example of even an hypothesised principle, which would greatly strengthen consistency arguments for the aspects of the framework associated with principles, is a grave weakness to the work of much of this chapter. Clearly presenting a simple example of an engineering principle of formal language design would be of value in any extension of the work.

An hypothesised principle would also aid the demonstration of the potential use of the principles aspects of the framework in the design of complex systems, and thus be an aid to demonstrating the external synthesis at the level of the framework. The discussions of Chapters 2 and 3 provide the only arguments of this potential at present.

The concept of the framework also draws indirectly on the work of others outlined in Chapter 2 on the nature of formality, as well as Chapter 3, on the nature of frameworks. The concept of applying category theory to design frameworks follows from Ho's work on Architectural Planning [Ho 82]. The notion of the distinction between craft, applied science and engineering conceptions of design is based upon the work of Long and Dowell [Long & Dowell 89], as is the notion of an engineering principle that is defined, operationalised, generalised and tested with respect to the problem of design. Thus, the aspects of the framework concerned with principles represent an increment of knowledge from fields as different as the philosophy of mathematics and architectural planning, and the philosophy of science and software engineering. Although this increment is largely conjectural, it may be of much use in resolving the problems concerned with the nature of knowledge for the design of complex systems.
12 Conclusion

The introduction presented in Chapter 1 outlined the criteria by which this thesis should be judged. Although incrementing knowledge was considered a part of the work of the thesis, it was pointed out that the primary aim was to create a synthesis of work across a broad range of disciplines. Thus, the four criteria that were outlined were as follows:

- Internal consistency between conception and framework
- External synthesis at the level of the conception
- External synthesis at the level of the framework
- Knowledge incremented at the level of the framework

This conclusion evaluates the thesis, considering each of these criteria in turn.

12.1 Conception - Framework Internal Consistency

In Chapter 1 of the thesis, assessing the internal consistency between the conception and the framework was said to require an answer to the question:

How well does the formal framework outlined in the second part of the thesis instantiate the conception presented in the first part?

The conception of formal language design was presented in Chapter 4. The basis of the framework was outlined in Chapter 5. The key chapters involved in instantiating the conception were: Chapter 6 which instantiated the notion of Requirements for formal language worksystems; Chapter 8 which instantiated the notion of the formal language worksystem artifact; Chapter 9 which considered the language syntax aspects of a formal language worksystem in more detail; and, finally, Chapter 11 which dealt with the notion of Formal Engineering Principles. The formal definition of Chapters 6, 8, 9 & 11 were directly derived from the conception in Chapter 4, thus demonstrating internal consistency between the conception and framework.

Nevertheless, problems remain, since the description of formal language worksystems, the artifact of formal language engineering, introduced in Chapter 8, was limited in its generality. Thus, although evidence has been
provided of the frameworks consistency, questions remain over its completeness. Discussions of how the framework might be completed and implications for consistency were discussed in Section 8.3.

12.2 External Synthesis at the level of the Conception

In Chapter 1 of the thesis, assessing the external synthesis at the level of the conception was said to require an answer to the question:

How useful is the conception as an approach to the design of complex systems?

This question can be asked both at the level of the generic engineering conception and at the level of the conception as instantiated for formal language design.

Evidence of the usefulness of the conception is of two sorts. Analytic evidence is based upon consideration of other work concerned with design and formality. Empirical evidence attempts to show the utility of the conception, through its use in the expression and solution of design problems.

The thesis made an analytic argument for the generic conception by sampling from the field of the philosophy of science and combining certain philosophies of science with work in establishing a conception of HCI design. Further evidence of the utility of the generic conception was obtained by considering the light it shed upon some of the disagreements that have arisen in the field of Software Engineering.

The thesis made an analytic argument for the specific aspects of the conception of formal language engineering, by sampling from the field of the philosophy of mathematics and using the philosophy of constructive mathematics as the basis for the definition of formal language that forms the basis of the conception.

Empirical evidence, by its very nature, is associated with the specific conception of formal language engineering. However, evidence of the utility of the specific conception also provides evidence of the utility of the generic
engineering conception. Empirical evidence for the utility of the specific conception has been presented at two levels in the thesis. Thus, the paint shop example which instantiated both the Requirements and Artifact aspects of the specific conception, together with the Venn grammars work of Chapter 9, illustrates the potential utility of the specific conception. However, such examples, are oriented more to illustrating the concepts rather than providing exemplification of the application of the conception to problems of likely clients. To demonstrate the potential utility of the conception in solving the problems of likely clients, the Benefit Requirements for Air Traffic Control (Chapter 7) and the Syntax of QOC (Chapter 10) were presented.

To fully illustrate the utility of the conception required some exemplification of the notion of a principle, and indeed further, the application of a principle to solve a design problem.

12.3 External Synthesis at the level of the Framework

In Chapter 1 of the thesis, assessing the external synthesis at the level of the framework was said to require an answer to the question:

How useful is the framework in supporting the design of complex systems?

This question can be asked of the framework as a whole and of the individual components of the framework.

The previous section offered empirical evidence for the utility of the conception. The examples mentioned, since they exemplified the notions of the framework, also provide evidence of the utility of the framework and the individual components of the framework.

The specification of the Benefit Requirements of Air Traffic Control illustrated the potential utility of requirements defined using pre-orderings. This work, with its ability to express trade-offs between different requirements for a work system, is of potential utility, not just in the design of formal languages, but perhaps, also, in fields such as software engineering.
The specification of the Syntax of QOC, indicated the utility of the Generalised notion of Language Syntax outlined in Chapter 9. Potentially, this work enables the definition of languages in a wide variety of mediums and, thus, is of utility in designing any artifact which consists of agents using a language to communicate in order to achieve some sort of task.

As with the conception, the utility of the framework as a whole can only be properly judged when principles have been constructed and consistently employed to provide solutions to design problems.

12.4 Knowledge Incremented at the level of the Framework

In Chapter 1 of the thesis, assessing the knowledge incremented at the level of the framework was said to require an answer to the question:

How much do the individual components of the framework increment existing work?

Despite the fact that the main aim of this thesis was a synthesis across disciplines, there has still, nevertheless, been a considerable amount on incrementing of existing knowledge.

Category theory has previously been applied to both architectural design and planning, as well as software engineering. As far as is known, there has been no attempt to embed the constructs of category theory within a formal notation such as Z. This embedding provides the vocabulary, of category together with the structural mechanisms of Z, thus enabling the use of categorial constructs in requirement and artifact specifications. The presentation of category theory, through simple design oriented examples, rather than complex and abstract mathematical structures, also amounted to an incrementation of existing knowledge.

The work on requirements, based upon the notions of pre ordering, is on the whole a completely novel approach. Although some similar work has been reported in the field of architectural design and planning, the work of the thesis was originally developed completely independently. The work of the
thesis in this area may be considered as incrementing what little work that exists on post-modernist requirements specification.

Although the thesis failed to arrive at a generalised definition of the notion of language worksystem, what has been established, from the new definition of formal language in Chapter 2, through to the characterisation of the simple formal language worksystem in Chapter 9, is a potential new approach to the design of complex systems, which balances concern for the agents of the worksystem with concern for the languages through which these agents communicate. Thus, the work on formal language worksystems offers a potential increment to all work involved in the design of systems which consist of communicating agents.

The work on generalising the notion of syntax is perhaps the most traditional piece of research in the thesis. The incrementation is the generalised notion of syntax, the generality of which has been demonstrated across a number of existing formalisms. The potential of the generalised notion has also been demonstrated through its application to the specification of the grammar of QOC.

Finally, the work on the conception, and the work within the conception on principles, directly increments the Dowell and Long work on HCI design knowledge. It uses the philosophy of science to extend their approach to design disciplines in general, as well as advancing work on engineering principles.
References


Name Index

Alexander 99
Anaxagoras 20
Apostol 175
Arbib 98
Aristotle 14

Backhouse 97
Barthelmann 258
Barwise 72
Beeson 32
Bickerton 172
Bishop 32, 34
Bourbaki 33
Brouwer 33, 34
Burstall 98

Carroll 68
Comte 19, 46, 53

Dasgupta 70
Davis 31
Drews 258

Ehrig 98, 258
Eizenberg 256
Englefreit 258
Euclid 29
Fetzer 71, 75
Feyerabend 52, 53, 78
Field 174
Finkelstein 88
Frost 38
Futatsugi 99

Gips 98, 261
Godel 30
Goguen 39, 41, 172
Goldblatt 98
Golin 260
Gottler 258
Green 40, 41
Guttag 99

398
Hamilton 96, 97
Harel 325
Hayes 73, 76, 99
Hersch 31
Heyting 35, 38
Hilbert 33, 34
Ho 98, 173, 384
Hoare 11, 70, 99, 249
Hodges 38
Hopcroft 98, 254
Hughes 88
Hume 29

Jackson 88
Jones 73, 76, 88, 99

Kant 29
Klein 29
Koning 256
Kreowski 258
Kreyszig 175
Kuhn 50, 51, 53, 58, 61, 78

Lakatos 33, 48, 51, 53, 78
Leahey 19, 20, 21
Loomes 37
Lyotard 172

MacLane 98
MacLean 25, 96, 334
Manes 98
Martin-Löf 14, 97
Milner 99, 249
Mitchell 256

Nuseibeh 88

Open University 37

Pavlidis 258
Pierce' 40
Plato 18, 32
Popper 48
Protogoras 21
Randall 88
Reeves 97
Reiss 260
Rozenberg 258
Rydeheard 98

Salter 11, 174
Scott 98
Shapiro 88
Siddiqi 172
Socrates 21
Sommerville 99
Spivak 175
Spivey 88, 99, 109, 152
Stiny 98, 256, 257, 261
Stork 99
Stoy 98, 243
Strachey 98
Sufrin 99

Tennent 98
Turner 73

Ullman 98, 254

van Heijenoort 33

Woodcock 37
Wright 256

Yourdon 88
Schema Index

Aircraft 190
altitude 185
ATCActualFlight 201
ATCBasicActualFlight 198
ATCBeacon 187
ATCBenefit 222
ATCCollisionCondition 210
ATCDesiredBenefits 221
ATCDomain 219
ATCEnRouteCondition 212
ATCFixedConditions 207
ATCFlightPlan 192
ATCInsideSectorCondition 209
ATCPlannedExitCondition 211
ATCPredictedFlightCondition 214
ATCSafe 204
ATCSafety 204
ATCSafetyCondition 215
ATCScenario 197
ATCSector 193
ATCStartConditions 208
ATCTotalFlyingTimeCondition 217
ATCTotalFuelConsumptionCondition 216
ATCUnplannedExitCondition 213
ATCWorkStructures 218
ATCWorkStructuresAttributes 205

basicNonTerminals 297
BenefitRequirements 160

category 113, 117
CategoryOfFunctorsOver 139
CategoryOfPartialFunctionsOver 123
CategoryOfPreOrderingsOver 141
CategoryOfRelationsOver 125
CategoryOfTotalFunctionsOver 121
climbingAbilityMax 187
Co-Cone 147
co-cones 148
co-limit 149
CommutativeDiagram 116
Cone 143
cones 144
CostRequirements 165
CostStructures 164
Cuboid 182
Cylinder 183

Date 102
DateAndTime 103, 104
Derivation 300
derivationArrows 302
derivationCategory 303
Diagram 115
DiscreteCategory 118

FinalColours 160
first 109
FL1 228
FL1DenotationalSemantics 227
FL1NullStatement 227
FL1Syntax 227
FL2 229
FL2DenotationalSemantics 228
FL2NullStatement 228
FL2Syntax 228
Functor 134

Grammar 298
Graph 111

idFunctor 135
InitialColours 160
isAnUpperBoundOf 178
isEpicIn 129
isIsoIn 131
isMonicIn 127

languageDefinedBy 304
latitude 185
leastUpperBound 178
LGraphComponents 335
liesBetween_and_ 180
limit 145
longitude 185

Message 105, 106, 109

Names 276
NegativeReals 177
NTArrow 292
nTArrows 293
nTCategory 295
NTGraph 271
Reals 180
RealDifferentiableOn 186
RealDifferentiableOn 185
RealInterval 181
Realop 204
RelationsOverD 119

safeSectorThroughput 194
safeThroughput 220
second 109
SimpleFL 226
SimpleFLWorksystem 231
StructureClass 277
sum 179

TArrow 290
tArrows 291
tCategory 294
terminalStructures 279
text-based grammars 306
TextStrings 227
TGraph 270
TGraphComponents 270
TLGraph 336
TotalFunctionsOverD 120
TPicture 273
TPictureComponents 273
TShape 267
TShapeComponents 266
TString 263
TStringComponents 263
TVennDiagram 326
TVennDiagramComponents 326

WorkDomain 157

\leq 137
P 135
\leq 108
\times 151
General Index

absolutism 20, 45
actual performance 54
agent 41, 230
agent combinations 245
agent requirements specifications 85
air traffic control 174
algebraic approach to graph grammars 258
algebraic specification techniques 99
algorithm 72
anarchist theory of knowledge 52
applied science 55
applied science conception of HCI 57
architecture 256
array grammars 261
artifact 62, 63, 80, 355
attributes 101
auxiliary hypotheses 49
axiomatic description 107
axiomatic descriptions, generic over classes 109
axiomatic semantics 243

basic transformations 244
benefit ordering 83, 159, 351
benefit requirements 83, 155, 159, 167
binary numerals 255

categories 110, 112, 258
categories of categories 132, 138
categories, examples of 118
category of artifact specifications 366
category of partial functions 123
category of pre-orderings 140
category of relations 125
category of requirements specifications 364
category of total functions 121
category theory 98, 245
CCS 99, 249
class-instance distinction 14
class-instance relationships 350
client requirements 63, 71, 355, 356
co-cone 146, 247
co-equaliser 150
co-limit 141, 148
co-product 150
commutative diagram 115, 116
component conception 87
components 101
conception 22, 23
conception of formal language design 79
conception of HCI Engineering 89
cone 143
consistency 171
constructive type theory 97
constructivism 14, 32
cost ordering 83, 165, 351
cost requirements 83, 164, 165, 167
cost structures 164
craft 55
craft conception of HCI 56
crisis period 50
criteria for assessment 26
cSP 99, 249
degenerate research programmes 49
denotational semantics 226, 243
derivation 255, 298, 299
derivation arrows 301
derivation category 302
design practice exemplars 64
design research exemplars 65
desired benefits 83, 159, 351
desired cost 83, 165, 351
desired performance 54, 83, 94, 155, 167
diagram 110, 115, 245
disciplinary matrix 50, 62
discrete category 118
domain theory 98
effective work 54
effector agent 232
empirical derivation 63
empirical validation 63
empiricism 19, 45, 51
employment of language 85
engineering 55
engineering conception 79
engineering conception of HCI 59
engineering design, a categorial meta view 364
engineering principles 94, 349
epic 126, 128
epistemology 18, 45
equaliser 150
executable specification 73
formal derivation 64
formal design paradigm 70
formal framework, basis 94
formal framework, choice 96
formal language 42
formal language design 79
formal language worksystem 80, 225, 231, 352
formal methods in software engineering 70
formal verification 64
formalism 33
formality and design 70
framework 22, 24
framework category 367
functor 132, 133
functor composition 135
general artifact specification 66, 352
general problem 54
general requirements specification 66, 351, 352
generalising syntactic structure classes 276
generic conception of engineering design 61, 67, 75, 78
game 29
grammar 298
grammatical formalism 254
graph 110, 111
graph grammars 258, 270, 315
guidelines 57

hard core 48
HCI 54
heuristics 56
hyperedge replacement 258

identifiers 101
identity functor 135
implement and test 56
initial object 150
interactive worksystem 54
interpretation of language 85
intersecting conception 87
intersection 150
iso 126, 130

knowledge 54
knowledge, definition 55
knowledge, generalisation 55
knowledge, operationalisation 55
knowledge, testability 55
specific requirements specification 64, 94, 351, 353, 355
specification 70
specify and implement and test 57
specify then implement 59
sub-category 136
substantive principle 354, 359
symbolic generalisations 50
syntactic components 262
syntactic medium 254
syntactic structure 254, 262
syntactic structure class 226, 242, 276
syntax 226, 242, 253
syntax of QOC 334, 338
system of values 50

task 42
terminal 254
terminal arrows 290
terminal categories 290, 293
terminal object 150
terminal relationship 262, 276
text-based grammars 254, 263
the extended framework category 378
theory dependance of observation 46
total functions 120
tree grammars 261
types 101

uncertainty of inductively defined truth 46
union 150

validation of discipline knowledge 68
Venn grammars 325
Vienna development method (VDM) 99
visual programming languages 260

work benefit 81
work domain 83, 156
work structures 156
worksystem costs 81, 164
worksystem performance 82, 83

Z notation 99, 101