BARRY DWYER

SYSTEMS ANALYSIS AND SYNTHESIS

Bridging Computer Science and Information Technology

MORGAN KAUFMANN

Systems Analysis and Synthesis

Bridging Computer Science and Information Technology

Barry Dwyer

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an Imprint of Elsevier

Acquiring Editor: Todd Green
Editorial Project Manager: Amy Invernizzi
Project Manager: Mohanambal Natarajan
Designer: Maria Ines Cruz

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451 USA

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloging-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-805304-1

Contents

This page intentionally left blank

Preface

Why You Should Read This Book

This book is meant as the basis for a final-year course in *Computer Science* or *Information Technology*. Alternatively, it can be read — with benefit — by a graduate who has come face to face with a real systems analysis and design problem, or even a seasoned systems analyst looking for a fresh and more systematic approach.

Many *Computer Science* and *Information Technology* students graduate knowing a great deal about programming and computer technology without having any idea how to apply their knowledge to the real-life problems of a business. Some — like those who specialise in IT support — might never need to, but many others will be expected to write application programs that directly satisfy business requirements. This can be a daunting task for the newcomer; a modest-sized business can have a system of procedures that seem unbelievably complex to an outsider. Unfortunately, there is rarely anyone who can explain the system. It may have evolved over many years into a complex mixture of computer and manual processes. Senior management may know *what* the system is meant to do, but not know *how* it does it. Middle management may understand parts of the system well, while lacking understanding of the details. Functionaries may know *how* to do things, but have little idea *why* they do them.

Discovering what an existing system does is called 'systems analysis'. Creating a new system is usually called 'system design', but I prefer the term 'system synthesis' because there are methods and algorithms that can be used to *derive* a correct implementation from a system's requirements.

An *experienced* systems analyst will often proceed intuitively — but not necessarily correctly — to an outline of the implementation, then use that as a framework on which to hang the necessary details. There are two problems with this approach: it needs the experience that the beginner lacks, and it often leads to an ill-conceived result. Therefore, in this book, I have used a simple specification language to state the business requirements independently of their proposed implementation. More importantly, I have used powerful mathematical methods from graph theory to derive conceptual frameworks that reveal *every* correct implementation. The graphs help the analyst *make* decisions; they don't merely record decisions the analyst has already made. It is these graphical methods that I consider to be the most important part of the book.

Although I introduce aspects of systems analysis early in the book, its main emphasis is on system synthesis. Only after understanding the convoluted way requirements become embedded in a system's infrastructure can an analyst stand any chance of understanding what requirements an existing system embeds. Analysis is a kind of reverse engineering. One needs design skills to reverse the design process.

System design is a many-faceted subject, and involves topics that are not normally regarded as part of a *Computer Science* qualification. In particular, the chapters on the human interface, system dynamics, and project management are likely to confront a

computer science student with new and interesting material. In the early chapters, I have included a summary of ideas that I expect most readers will already know, but biased towards to the specific requirements of the overall subject.

Systems analysis has as many facets as there are types of business. If you want to create a computer accounting system, it is wise to read a book or two about accountancy, if you want to create a job scheduling system for a factory, it is wise to read a book or two about job scheduling, and so on. It is impossible to cover all such topics in detail. In fact, the only case study explained in detail concerns an imaginary educational institution. There are two reasons for this choice: the reader has almost certainly encountered a real one, and it is an application with which I am all too familiar. Even so, the case study has been scaled down to be much simpler than any real system — otherwise the book would have been longer without being any more instructive.

I debated for a long time whether the title of this book should use the phrase "information system" or simply the word "system", because I would argue that every interesting computer system is an information system. Even a micro-controller for a production line is a primitive information system. It knows certain things, such as what product is to be made, and the current availability of parts. It responds to human input through an interface — perhaps as simple as a *Start–Stop* button. It acts in a planned manner, and ultimately it causes useful things to happen. These are all defining characteristics of an information system.

Credentials

After graduating in physics, I began my career as an electronics engineer in the aerospace industry, designing flight simulators for guided missiles and learning about flight control systems. At that time, I could truly claim to be a rocket scientist. I then moved on to designing hardware for head-up displays. During this time I learnt a lot about the human interface as it concerns the pilot of an aircraft.

Finding that I could (at that time) earn more by writing software than by designing hardware, I moved into software engineering — usually concerning file and database systems, but with occasional forays into application programming when more interesting work was unavailable. My last job in London was designing specialised database software for the first computer criminal records system at Scotland Yard.

In the early 1970's, I migrated from England to Australia, where I was in charge of software development for the administration of a tertiary education institution, setting up their first interactive computer systems for book-keeping and student registration.

After a while, I drifted into teaching, and for a period of ten years or so I taught a one-semester course called *Systems Analysis and Project* at the University of Adelaide: Final-year students were grouped into teams of five or six, and each team would find five or six potential clients. An assistant lecturer, Heiko Maurer, and I would select the projects that we thought most practical for the teams to design *and implement* within the rather limited time frame. Clearly, my own experiences as a systems analyst and computer programmer were a help. Heiko and I acted

as design consultants, and were vicariously exposed to a wide range of systems analysis problems. The projects proved amazingly varied, ranging from machine-shop scheduling to matching the dental records of fire victims. Almost all of them involved designing a database. We found that students always underestimated the number of data structures they would need, wrongly believing that one or two complex structures would lead to a simpler implementation than a larger number of simple ones. The book therefore devotes a lot of space to database design.

Students were required to hold weekly project meetings. The minutes of these meetings were posted to an on-line database, and the jobs they had assigned one another were displayed by a utility program as a Gantt chart. Alongside the practical work, students received twice-weekly lectures, in which I aimed to deliver the information they needed in time for them to use it.

During the course of a single semester, students would progress from a state of complete bewilderment to one of pride in a job well done. Several groups went on to exploit their ideas commercially.

When I worked as an application programmer, I was convinced that all my problems were caused by systems analysts. When I worked as a systems analyst, I learnt that the problems were instead caused by the *infrastructure* upon which applications were built — be it batch processing, client/server, Internet, or whatever. My own experiments have satisfied me that under 10% of application code may have anything to do with the business rules; the rest is needed to interface with the user and the infrastructure.

During my time, I have seen digital computers increase tenfold in speed and storage capacity every decade — meanwhile steadily decreasing in size and cost. I have worked with hardware, software, and humans in a wide range of applications, and at the end of it, I am convinced that *all* systems are united by a few common principles, and that these principles can be taught.

Plan of the Book

The chapters in this book roughly follow the order of the *Systems Analysis and Project* lectures I gave at the University of Adelaide. For the reader's benefit, the first three chapters include background material that my students were assumed to know, slanting it in a way that is relevant to later chapters. These early chapters condense the contents of a good sized textbook, so the treatment is necessarily sketchy. The reader who is unfamiliar with this material is advised to consult the texts mentioned in the *Further Reading* sections at the end of the chapters concerned.

The later chapters contain material that I could only have included in a full-year course, and every chapter goes into rather more depth than was possible in lectures. Despite this, the chapters remain summaries of material drawn from several sources, and most chapters — and indeed some sections — could fill a textbook. The interested reader is again invited to study the *Further Reading* sections.

- Chapter 1 describes the scope of the book, explains the difference between analysis and *synthesis*, and points out that intractable methods of system design are potentially useless.
- Chapter 2 presents the mathematical background to the ideas from *graph theory* that dominate the book. It also outlines the many ways these mathematical notions can be represented within a computer system.
- Chapter 3 concerns *atoms* and *events*, the fundamental building blocks of systems. Atoms are modelled by finite-state automata that change state in response to events. The chapter also discusses the relationship between states and behaviours.
- Chapter 4 shows how to relate the different kinds of atoms with *functional dependencies*. It is a *graphical approach to database design* that immediately results in a well-designed schema.
- Chapter 5 introduces the reader to a *specification language* designed to describe the business rules that define events — in a way that is independent of their implementation.
- Chapter 6 discusses the way *database system*s work. This is useful not only to a reader who uses a commercial database management system, but also to one who must hand-craft data structures for some other infrastructure. It also introduces the problem of *deadlock* in concurrent processing systems.
- Chapter 7 examines how events control the way data flow between processes, and how *use-definition analysis* can affect system synthesis.
- Chapter 8 concerns the *interfaces* between a system and its environment, especially the *psychological factors* concerning its interfaces with humans.
- Chapter 9 describes how *business rules* can be expressed in ways that are more easily understood and modified than expressing them directly as program code. It includes a discussion of *expert systems*.
- Chapter 10 discusses the dynamic behaviour of systems, including *queueing theory*, modelling and *simulation*.
- Chapter 11 discusses *project management* and control. It culminates with a section that describes how the various tools discussed in earlier chapters can be put together as a successful *methodology* for system development.
- Appendix A explores the relationship between state-based and grammar-based descriptions of system behaviour.
- Appendix B discusses the traditional normalisation approach to database design — if only because its terminology is required knowledge for anyone who uses databases.
- Appendix C gives the answers to the exercises at the end of each chapter.
- The book ends with a comprehensive index.

The chapters are arranged in an order that develops ideas and tools in a logical progression. This may not always be the order in which the tools need to be used in a typical project. In particular, in teaching a project-based one-semester course it will prove necessary to take things in whatever order is needed to complete projects on time. In this context, a preliminary feasibility study would be a luxury; students

will first need to know how to structure data, how to design interfaces, and how to schedule their own activities. Once they have made a start on their projects, lectures can fill in the details — in much the same order as the chapters in this book. The content of such lectures will depend on what students can be assumed to already know. For example, the students I taught were already familiar with *SQL* and finite-state automata, among other things.

Boldface type is used in this book whenever a new concept is introduced and defined, either formally or informally. Such concepts appear in the index with the page number of the primary reference set in boldface type. Boldface is also used for programming language keywords. Italics are used for names of all kinds, and sometimes, just for *emphasis*.

This page intentionally left blank

Acknowledgments

My thanks go to:

- Sid Deards, my old Circuit Theory lecturer, who first showed me the advantages of synthesis over analysis,
- Paddy Doyle, who first showed me that synthesis works for software too,
- Michael A. Jackson, my mentor in the subjects of software development and project control,
- Heiko Maurer for his help with all my student projects,
- my ex-students, for teaching me how to teach,
- Carole Dunn, Peter Perry and David Knight, who have read drafts of various chapters and told me where I could improve them,
- the various implementors of LaTeXand of *Prolog*, without whom I would never have got this book into shape,
- my wife, Linda, for her patience, loyalty and tolerance.

This page intentionally left blank

List of Figures

List of Tables

This page intentionally left blank

List of Algorithms

This page intentionally left blank

CHAPTER

Systems, analysis, and synthesis

1

CHAPTER CONTENTS

1.1 SYSTEMS

A **system** is a collection of parts, or components, that interact in such a way that the whole is more than the sum of its parts. For example, a bicycle is a system, but if you buy a kit to make a bicycle, you don't yet have something you can ride. You don't have a system. First, you must assemble the parts. Correctly. The connections and interactions between the parts are just as important as the parts themselves.

This book is not about bicycles, but about systems, mainly information systems. Even so, there are useful analogies between information systems and physical systems such as bicycles, simply because they are both examples of systems. Further, the function of an information system is usually to model some kind of physical system, so that the two become analogues of one another. For example, a stock control system models the movement of physical stock in and out of a store, and an accounting system models the movement of money between accounts. (Indeed, the

Systems Analysis and Synthesis: Bridging Computer Science and Information Technology. DOI: 10.1016/B978-0-12-805304-1.00010-2

modelling of money has become so well established that actual money is involved less and less as time goes on. The miser no longer has to count his gold. The shopper carries a credit card instead of a purse. Even coins and notes are mere symbols for the gold and silver bullion they replaced.) So, by information system, we mean something that models aspects of the real world. In principle, the medium it uses to do this is unimportant, but we are primarily concerned with the medium of the digital computer.

Some systems are obviously hybrids of a physical system and an information system. For example, a robot, or a guided missile, has eyes and limbs that are physical sensors and actuators, but they are interconnected by an information system. Less obviously, what we may think of as a 'pure' information system usually has eyes and limbs that belong to humans. Data are entered into a stock control system because a store hand perceives that goods have been delivered. When the stock control system issues a purchase order, it is up to a human, somewhere, to supply some physical goods to replace those that have been used.

What makes an information system relevant to the real world is that both must obey the same *rules* or *laws*. Among the most fundamental physical laws is the conservation of matter.[1] Every physical object that enters a stock control system should either eventually issue from it or remain in stock. Money can move between accounts, but a good accounting system cannot allow it to evaporate. If it turns out that the model and the physical system disagree, we suspect wrongdoing.

Using computers to model systems is not a one-way street. If you want to explain a computer sorting algorithm to someone, you may decide to demonstrate it with a deck of playing cards. This suggests that the study of physical systems can enlighten our understanding of information systems, and *vice versa*.

Many books on system design suggest that every software component should have a single well-defined function. Despite this, we mustn't assume that this is a property of *efficient* systems. A bike tyre is part of its propulsion system, part of its steering system, and part of its braking system. It also contributes to the comfort of the rider. It is *possible* to make systems in which each part has a single well-defined task, but it isn't always a good idea. A bike rider *could* lower a skid in order to stop, which would mean that the tyres could be better designed for their other functions, but we don't make bicycles that way.

Design decisions are not universal. If we consider aircraft technology, tyres don't contribute to propulsion in any way, and they contribute to braking and steering only during taxiing. At higher speeds, reverse thrusters or parachutes provide the brakes, and control surfaces do the steering.

Why are systems made of components at all, rather than in one piece? In the case of the bicycle, one reason is forced on us: some parts, such as the wheels, have to move relative to other parts. Why isn't the tyre part of the rim of the wheel? Because it is made using a different technology. Why isn't the rear axle part of the frame? First, because it would be impossible (or at least, very difficult) to install the wheels. Second, because the axle is best considered as part of the wheel. Why? Because its

[1] Strictly speaking, 'matter and energy'.

interface with the wheel is more complex than its interface with the frame. Keeping the interfaces between parts simple is an important aspect of system design. Given simple interfaces, it then becomes possible to have different people design and make the parts. For example, a bicycle's wheels may come from a specialist supplier.

These same considerations apply to information systems. Consider a system that allows a user to browse a database across the Internet. Because the user is in one place and the database is in another, the system is forced to have at least two parts: a client and a server. Typically, the client process will use a web browser driven by *HTML*. On the other hand, since the server has to access a database but must also use Internet protocols, it will probably use a combination of *SQL* and *HTML* technologies, and although *SQL* and *HTML* will need to interface with one another (perhaps using *PHP*), they will tend to form separate components because they use different technologies. The *SQL* and *HTML* aspects of the system may well be implemented by different specialists. Making a system from components makes it easier to put the system together. A good designer makes a system of parts that have simple interfaces. This is where a system designer has the most freedom — but has to make difficult decisions. Once these decisions are made, the rest is routine.

The design of efficient systems depends on the current technology, and technology is always changing. A danger facing all system designers is the failure to adapt to technology change, and to continue to use methods that have become outdated. When we see old movies of the original Wright brothers' *Flyer*, we see a machine whose construction more closely resembles that of a bicycle or a kite than of a modern aircraft. We cannot criticise the Wright brothers for this, but we would certainly sneer at a modern airliner that used such out-dated technology.

Despite constant technological change, anyone who has designed systems for any length of time realises that some aspects of past experience *are* transferable to new technologies, and that *some* aspects of system design must therefore remain constant. In this book, we attempt to isolate and teach these fixed principles, as applied to information systems.

An important feature of man-made systems is that they are designed to be useful to *people*, even if it is merely to entertain them.[2] Therefore, anyone who wants to make good systems has to understand people. This area of human-machine interaction is known as 'human factors', or 'ergonomics'. Again, there are principles that govern ergonomics, which this book aims to teach.

1.2 AIM OF THE BOOK

Systems analysis usually concerns information systems, mainly in business. These systems are often complex because of their size or because of their intricacy (i.e., having many processing rules), rather than on account of tricky or time-consuming algorithms.

[2] Indeed, the notion that systems have a purpose is so central to our thinking that we tend to assume that *all* systems have one. We look at the universe and ask, 'What is it all for?'

Systems analysis includes everything from identifying a problem to implementing its solution, except routine programming tasks. The output of the analysis must present no technical challenge to the programmers who will implement the system. Few activities in systems analysis can be formalised as algorithms; most rely on experience or intuition. Once something can be formalised, it tends to be incorporated in the next high-level language or CASE (Computer-Aided Software Engineering) tool. It ceases to be considered as part of systems analysis, which therefore consists mostly of informal methods.

In this book, we present a new approach to the design of systems, with an emphasis on information systems. Despite this emphasis, the techniques taught here are general enough to apply to many other problem areas.

The approach is new in two ways:

First, there are many ways a set of operational requirements can be implemented by a physical system. Here, we express aspects of the operational requirements in a way that is independent of the implementation. The result is a formal specification that allows us to derive systems that are provably correct. They will do what the user wants. Other aspects are less easily formalised, and concern how the system looks and feels, and the technology that underlies it. We discuss these other aspects, but aren't (yet) able to formalise them.

Second, the approach focusses on what we call atoms, a concept similar to objects (as in object-oriented programming). Atoms differ from objects in that they are usually more primitive and don't necessarily map directly onto program objects or other program structures. In other words, there may be an intermediate stage of design between atoms and their expression as objects. The approach is not incompatible with object-oriented design, but it sees it as one of many alternatives.

1.2.1 KERNEL, INTERFACE, AND ENVIRONMENT

Every system operates in an environment. It connects to the environment through interfaces. For example, a bicycle has interfaces with its rider, the road, and the air, which together form its environment. (In a larger sense, its environment includes the road system and its rules and regulations.) Its interface with the rider includes the handlebars, pedals, and saddle. Its interface with the road is its tyres. Its interface with the air is its aerodynamic shape, which is one reason why racing bikes have lowered handlebars.

As time goes on, interfaces tend to become more sophisticated. For example, racing cyclists wear special shoes to fit the pedals, and skin-tight costumes to reduce aerodynamic drag.[3] Even so, the core purpose of the bicycle remains constant: to provide a speedy means of man-powered surface transport. This purpose is the kernel of the system.

[3] The international rules of bicycle racing limit the aerodynamic sophistication that is allowed, otherwise racing bikes would reach even more dangerous speeds.

Consider a typical business system. In the distant past, customer orders would have been received by letter, transactions recorded in a journal, information stored in ledgers, and bills sent by mail. In the early days of computing, orders might have been punched onto cards, stored in computer files, and bills printed by a line printer. Somewhat later, the orders might have been typed into a text-based 24-line monitor, stored in a relational database, and bills printed by a matrix printer. Then would have come a mouse-driven graphical interface, with orders received and bills sent by e-mail, then a touch-driven or voice-driven interface, and, in the future, perhaps a thought-controlled three-dimensional virtual-reality experience.

Historically, each change of the interfaces meant that most of the software had to be rewritten. The interfaces are simply[4] an application of current technology. In practice, over 90% of the programming task is concerned with interfaces, either with the environment or with a database. The point is, throughout all these changes, the kernel remained the same: to bill customers for the goods they ordered. The kernel is the set of **business rules**. A specification of a system needs to describe its rules in a formal notation, such as mathematics or a programming language. The interfaces are often best described using pictures or through design standards.

This doesn't mean that interfaces are unimportant. Badly designed interfaces are frustrating to use. Well designed interfaces are fun and rewarding.[5] Fortunately there are some psychological principles that we can exploit to get them right.

1.2.2 SYSTEM DEVELOPMENT MODELS

There are at least three traditional models of system development:

- Classical **life-cycle models** divide system development into a series of well-defined phases.
- **Evolutionary models** (or **prototype models**) develop a system by successive approximation.
- **Transformational models** develop a complex system by transforming a simpler one. This book will focus on a particular transformational model.

All methods involve the same kinds of activity:

Feasibility Study: Deciding which problems are worth solving.

Requirements Analysis: Deciding exactly what the problem is.

Interface Design: Deciding how the system will look to users.

Simulation and Modelling: Exploring the behaviour of a proposed system *before* building it.

[4] Simple, but very time-consuming.

[5] I'm sure you can think of both good and bad interfaces that you have used.

Data Design: Deciding how data will be represented and stored.

Process Design: Deciding what processes will be needed and how they should work.

The first four steps are usually performed in consultation with a client. In large organisations these activities are usually carried out by a team, often with a rigid job hierarchy, passing system documentation down a chain from senior analyst to programmer. In smaller or more democratic organisations, one person may perform the whole job, from feasibility study to programming.

Systems analysts are often ex-programmers, ideally with expert knowledge of the problem area. If that is lacking, **design methodologies** can help the analyst come to grips with unfamiliar problems.

1.2.3 CLASSICAL LIFE-CYCLE MODELS

A typical classical methodology is to divide activity into five phases:

- Requirements Analysis,
- Logical Design,
- Physical Design,
- Implementation,
- Maintenance.

These phases are separated by contracts with the client, based on written specifications. Since the specifications cascade from phase to phase, this approach is also called a **waterfall model**.

At the end of the **requirements analysis** phase, the specification may say what activities the system is to perform, what the benefits of the new system will be, what existing procedures are to be replaced, and how much the system should cost. The client will then check this document and decide whether to proceed with the next phase.

At the end of the **logical design** phase, the client may receive a draft operations manual and revised estimates of how much the system will cost. Again, the client will decide whether to approve further progress.

Physical design has little impact on the client, except in refining the documentation from the logical design. It is mainly directed at solving implementation problems.

At the end of the **implementation** phase, the client will receive a working system, and instructions for using it. Programs will also be documented in preparation for maintenance.

Once the system is installed, it will need to be *maintained* for two major reasons: to correct deficiencies in its design (e.g., bugs), and to adapt to changing business requirements (e.g., changes in tax laws). **Maintenance** accounts for about 75% of all the work done by information systems departments. This may not be a bad thing and may simply reflect the robustness of the original physical design.

Various life-cycle models are broadly similar, but differ in the number of their phases, and the precise documents that define the boundaries between them.

All life-cycle models have drawbacks:

- Divisions between phases are rarely clean. For example, problems may be found during the implementation phase that affect the feasibility of the system or cause the physical design to be changed.
- Clients don't understand what they are agreeing to; even a computer expert finds it hard to understand a complex system from its documentation. Few people have enough imagination to foresee how a complex system will work in practice, especially if the technology is new or unfamiliar.
- It is almost impossible to keep all the documentation up to date, especially once software maintenance begins.

1.2.4 EVOLUTIONARY MODELS

Evolutionary models are developed by building a crude version of the system, which users can then criticise. This feedback is used to improve the design step by step, until it is satisfactory. This approach has been made easier through the use of '4th generation' systems products, often based on a database management system, which allow much more **rapid prototyping** of application programs than is possible in procedural languages. One advantage is that users may start using an incomplete system before its 'luxury' features are implemented.

Evolutionary models have some drawbacks too:

- The first hastily constructed prototype should probably be thrown away, but programmers are reluctant to discard anything in which they have invested time and energy.
- A smooth evolution from the user's point of view may not correspond to a smooth evolution of the system design. There is a tendency to adopt quick fixes, rather than long-term solutions, until the system becomes incomprehensible, and further evolution becomes impossible.

1.2.5 TRANSFORMATIONAL MODELS

Transformational models are based on the view that most of the complexity in systems arises from the need to make them efficient. If efficiency didn't matter, much simpler designs would often be possible. The transformational approach is to design the simple system (a logical design) and *transform* it into the desired system (a physical design). This transformation can be achieved either by hand, through programming tools, or (ideally) completely automatically.

- In the case that the transformations can be both made and *chosen* automatically, the methodology often becomes formalised as a feature of a programming system.
- In a 'strong' methodology, an algorithm makes the transformations, although the designer must choose the correct set of transformations intuitively.

- In a 'weak' transform methodology, the transformations themselves must be made intuitively, but their correctness can be checked formally.
- A methodology in which the correctness of a step cannot be checked is called 'informal'.

Transformational models often evolve from informal techniques, by way of formal techniques, to programming tools. They are areas of active research.

1.3 ANALYSIS

The word 'analyse' means to separate something into its parts in order to understand it better. This especially includes, of course, the way the parts are interconnected. Systems analysis can therefore be similar to reverse engineering — but without the suggestion of commercial espionage. We are given some existing system, and we want to know how it works, usually with the objective of making a new system to do the same things more cheaply or efficiently.

There is danger here: we can sometimes see *what* something does without knowing *why* it does it. We can analyse a bicycle and deduce that the pedals turn a gear wheel that powers a chain that turns a sprocket that turns a wheel. But it is only after we learn that it is not an exercise machine, but a vehicle whose wheels go on the ground, that we have any chance of improving its design, and perhaps inventing the motorbike.

A systems analyst is therefore someone who studies a system in order to understand it, usually with the purpose of improving it. The analyst has an advantage if he or she has expertise in the field in which the system is used. For example, to understand a business accounting system it is well to have had a few lessons in accounting. Although it is sometimes possible to learn about the relevant field by observing the system itself and asking intelligent questions about it, this has three disadvantages:

- The existing system may already be dysfunctional and a poor example to follow.
- It is often easier to find out what the system should do from a textbook.
- The system may do some things that are no longer, or never were, useful.

I am reminded here of an acquaintance who was asked to design a computer system for an insurance company. He wasn't the first to be given that task. Several others had tried and failed. The inner workings of the existing office systems were so complex that no one had managed to understand them. Realising this, he refused to even think about the existing system and spoke only to the company executives and the sales representatives — the people at the top and the people at the bottom — the company's interfaces with the outside world. He went on to succeed where the others had failed.

The reason is plain. He didn't ask 'How?', but 'What?' and 'Why?' The 'How' was related to the technology of filing cabinets and forms-in-triplicate and had evolved into a system of baroque complexity. (For a likely reason, see 7.5.3.) Given

computer technology, the existing 'How' became irrelevant. By knowing 'What' and 'Why', inventing the new 'How' became straightforward.

Given that the likely reason for analysing a system is to improve it, a good systems analyst should be familiar with the latest technologies for producing a new system. Too often analysts specialise in one technology, say a particular database system, and use their favourite technology irrespective of the problem. As the saying goes, 'To a hammer, everything looks like a nail'. New technologies are always emerging, and the wise analyst embraces the new, without necessarily rejecting the old.

'Analysis' has a second meaning. We can also analyse by making experiments. These experiments can take place on the actual system being studied or on a model of it.

In an input-output experiment, we can analyse the behaviour of the system without examining how it works. This is how we might learn how to use an unfamiliar piece of software for which we have no documentation. Unfortunately, we can never be certain that we have discovered all its features. We may even invent our own theory to explain to ourselves how it works, but we shouldn't be too surprised when our theory proves faulty.[6]

A second kind of analysis is to build a model. We use models for a variety of reasons. One is that we want to study a system before going to the expense of building it. For example, we may want to assess the quality of the user interface, perhaps simply by making mock-ups of reports or screen-based forms and discussing these with users. We may want to investigate the system's ability to handle heavy workloads by building a simulation of its queues and servers; or, if the system includes some measure of decision-making, or artificial intelligence, we may want to check that it chooses wisely.

An interesting aspect of models is that the same model can describe systems that are widely different physically. For example, in a later chapter we shall find surprising similarities between the mathematical models of a stock control system and of a person using a shower. The advantage of such similarities is that, arguing by analogy, if we can learn to control the shower, we can learn to control stock.

1.4 SYNTHESIS

Many books in this field have a title such as 'Systems Analysis and Design'. Why does the title of this book use the word 'synthesis' rather than 'design'?

An electronics engineer designing an active linear filter knows that such a filter can only be made if its behaviour can be described by the ratio of two polynomials of a certain form. If it can, the engineer can derive two three-terminal networks and connect them in a certain way to an operational amplifier, and the design problem is solved. The details and the jargon aren't important to us; the point is that if the problem can be expressed in a certain way, then it is routine to design a filter that does the job. The resulting filter is correct by design. It is almost a no-brainer. A

[6] Scientific theories are like this.

computer can do it. On the other hand, if the requirement can't be expressed as a ratio of polynomials, no linear filter can possibly do the job — period. The engineer must either approximate the requirement as such a ratio, or use a different technology.

Let us look at an example of synthesis relevant to information systems:

Imagine that we have the following business rules:

Rule 1: If $balance \geq debit$ and $status$ = 'ok' then
set $approve$ to 'yes' and $error$ to 'no'

Rule 2: If $balance < debit$ and $status$ = 'ok' then
set $approve$ to 'no' and $error$ to 'no'

Rule 3: If $status \neq$ 'ok' then
set $error$ to 'yes'

By some intuitive process, we might *design* the following snippet of program code,

```
if (status == 'ok') then {
  if (balance < debit) then
    approve = 'no'
  else
    approve = 'yes';
  error = 'no' }
else
  error='yes';
```

How can we know that this procedure is correct? In this case, it is not hard to prove: we can check that each rule is implemented properly. In general however, especially if a program contains loops, it can be hard to prove that it is correct.

The alternative approach is to *synthesise* the procedure. Algorithm 1.1 is suitable for sets of rules of this type:

Algorithm 1.1 Converting rules into procedures.

Step 1: While the last action in each rule is the same, place it after all the rules, and eliminate it from further consideration.

Step 2: Pick some condition. Divide the rules into two groups: those for which the condition is true and those for which it is false. If a rule does not specify the condition, put it in both groups.

Step 3: Generate a conditional **if**-statement whose *true* branch contains the first group of rules and whose *false* branch contains the second group of rules.

Step 4: Apply the same procedure to each group of rules recursively, but exit the recursion when a group of rules becomes empty.

Let's see what happens to the example:

Step 1 The last actions differ; nothing happens.

Step 2 We pick the condition (*balance* < *debit*).
Group 1 contains,

Rule 2: if *status* = 'ok' then set *approve* to 'no' and set *error* to 'no'.

Rule 3: if *status* ≠ 'ok' then set *error* to 'yes'.

Group 2 contains,

Rule 1: if *status* = 'ok' then set *approve* to 'yes' and set *error* to 'no'.

Rule 3: if *status* ≠ 'ok' then set *error* to 'yes'.

Step 3: We generate,

```
if (balance < debit) then
    {Group 1}
else
    {Group 2};
```

Step 4: Applying the same process recursively to Group 1 we have,

Step 1 Nothing happens.

Step 2 The only condition is (*status* = 'ok').
Group 1.1 contains

Rule 2: set *approve* to 'no' and set *error* to 'no'.

Group 1.2 contains

Rule 3: set *error* to 'yes'.

Step 3 We generate,

```
if (status == 'ok') then
    {Group 1.1}
else
    {Group 1.2};
```

Step 4 Applied recursively to Group 1.1 we have

Step 1 The last actions agree; we generate `error = 'no'` (to be last) and delete "set *error* to 'no'," then generate `approve = 'no'` and delete "set *approve* to 'no'."

Step 2 Nothing to do.

Step 3 Nothing to do.

Step 4 Exit the recursion.

and so on . . .

At the end of this process we have,

```
if (balance < debit) then
    {if (status == 'ok') then
        {approve = 'no'; error = 'no'}
    else
        {error = 'yes'}}
else
    {if (status == 'ok') then
        {approve = 'yes'; error = 'no'}
    else
        {error = 'yes'}};
```

How do we know that the resulting procedure is correct? Because somebody has already proved once and for all that if Algorithm 1.1 is followed, the solution is *always* correct.[7]

You will notice that the synthetic solution is longer than the one that was derived informally. On the other hand, if we had picked the two conditions in the reverse order, we would have reached the same solution as the informal approach. Unfortunately, although two conditions can be placed in only two possible orders, as we increase the number of conditions, the number of their possible orderings grows very rapidly.

This example shows an important property of synthetic methods. It can be easy to synthesise solutions that are guaranteed to be correct, but very many solutions may be possible, and it may be hard to choose the best. Therefore, for synthesis methods to be practical, we need also to have some ways of finding optimum or near-optimum solutions.

1.4.1 COMPOSITION AND DECOMPOSITION

There is a subtler reason for using the word 'synthesis' in the title of this book, which concerns the way we approach the question of optimisation. How do we decide from what parts a system should be made? Traditional approaches usually consider how best to decompose a system into parts. This is the direction of analysis: breaking things down. The alternative is to first split the system into as many fragments as possible (atoms), then to decide how best to compose them together. This is the direction of synthesis: building things up.

The reason why composition can be better than decomposition is that when we take a system as a whole and separate off any small fragment, the system almost certainly gets worse. We need to decompose a system into larger parts before we can expect it to improve. Guessing how to do this is often something that only comes with experience, and experience isn't an easy thing to teach, either to a human or to a computer. On the other hand, when we compose two small fragments into a single

[7] A program to convert rules to program code in this way would need to diagnose errors in the set of rules, e.g., when combinations of conditions exist where no rule can be applied.

Table 1.1 The time taken to generate the permutations of a list of numbers overtakes the time taken to sort it as soon as it contains four elements

Length	1	2	3	4	5	6	7	8	9
Sorting	1	4	9	16	25	36	49	64	81
Permutations	1	2	6	24	120	720	5,040	40,320	362,880

component, we often see an immediate improvement. We simply let the components grow. This is an approach we can teach. Even a computer can do it.

1.5 TRACTABILITY

In this section, we discuss the question of whether algorithms have any practical value. An algorithm is quite useless if it takes too long. For example, an instruction such as, 'Assemble the basic parts into components in the way that maximises efficiency', is quite useless, even if the measure of efficiency is well defined. The reason is that there are typically so many ways of assembling things that we could never consider them all. If a system has only 10 basic parts, they might be combined in over 1,000 ways. If it has 100 basic parts, they might be combined in more than 10^{30} ways.[8] It will be our aim to avoid such useless advice.

Formally, we speak of some problems being tractable and others being intractable. These terms refer to how rapidly the time or storage space needed to solve a problem grows with the size of its input. For example, even using a simple method such as bubble-sort, a list of names can be sorted in time proportional to the square of the length of the list, and more advanced algorithms such as Quicksort can do even better. On the other hand, if we are asked to generate all the permutations of a list of names, the time needed grows much more rapidly, in proportion to $n!$, the factorial of its length, n — simply because that is how many permutations it has.[9]

As Table 1.1 shows, the time taken to sort is overtaken by the time taken to generate permutations as soon as we reach four items. Of course, the algorithm for sorting might be more complicated than that for generating permutations; it might take 100 times longer to sort a list of one item than it does to generate its only permutation, so the comparison would no longer be fair. Table 1.2 shows what happens if we multiply all the sorting times by 100. Despite this change, the time taken to generate permutations quickly overtakes the time taken to sort the list, as soon as we reach seven items.

[8] In fact, in many more ways than that. These estimates simply assume that each part can be assigned to one of two subsystems. There can be three subsystems, four subsystems, and so on.

[9] $n! = (1 \times 2 \times 3 \times \cdots \times n)$.

Table 1.2 Even if the sorting algorithm is inherently slower by a factor of 100, the time taken to generate permutations overtakes sorting as soon as the list contains seven elements

Length	1	2	3	4	5	6	7	8	9
Sorting	100	400	900	1,600	2,500	3,600	4,900	6,400	8,100
Permutations	1	2	6	24	120	720	5,040	40,320	362,880

Another way to see the contrast is to imagine the effect of using a faster computer. Suppose that each unit in Table 1.2 represents one microsecond of CPU time for our current computer, so it can generate all the permutations of nine items in 362.88 milliseconds. In about the same time, it could sort a list of not just nine, but about 60 items. Now suppose a new computer is 100 times faster. Again, in the same time, it would be able to sort over ten times as many items, but it would be able to generate the permutations for a list with only two extra items. From this, we learn that there are problems that future technology will do little to help. We call these problems *intractable*, and the remaining problems *tractable*.

Letting n denote the size of the problem — usually the length of its data, Figure 1.1 shows the growth rates for several functions of n on a logarithmic scale. Multiplication by a constant (e.g., 100) merely displaces a line vertically. The plots for n, n^2, n^3, and so on are straight lines of increasing slope. Because of this, the highest power of n will always dominate. For example, the function $f(n) = 100n + n^2$ is dominated by n^2: once $n > 100$, $f(n) < 2n^2$.

In **complexity theory**, we make a clean division between tractable and intractable problems. If the time or storage space needed to solve a problem is never greater than some *finite* polynomial expression involving n, the problem is called **polynomial**, and is considered **tractable**.[10] If no such polynomial exists, it is considered **intractable**, and is often loosely referred to as **exponential**.[11] The distinction between tractable and intractable problems doesn't depend on constant factors (such as the factor of 100 in the sorting example) because, above a certain value of n, an exponential algorithm will always take longer than a polynomial one. An intractable problem is therefore one whose growth on a log-log scale cannot be bounded by a straight line.

We denote the **order of complexity** of an algorithm using **big-O notation**: an algorithm with execution time $f(n)$ has complexity $O(g(n))$ if $f(n) < kg(n)$ for some constant k, and sufficiently large n. Thus $100n + n^2$ has order $O(n^2)$ because $100n + n^2 < kn^2$ for $k = 2$ and $n > 100$.

[10] Although $\log_y n$ cannot be accurately expressed as a *finite* polynomial, since $\log n < n$ for all $y > 1$, a problem with complexity $O(\log n)$ is tractable.

[11] The log function is such that $x = \log_y n$ if $n = y^x$. Logarithmic growth is therefore the antithesis of exponential growth: in exponential growth, time grows exponentially with n; in logarithmic growth, n grows exponentially with time. In complexity theory we often omit to mention y. Its value only displaces the log-log curves for $n = y^x$ and $\log_y n$ by a constant factor.

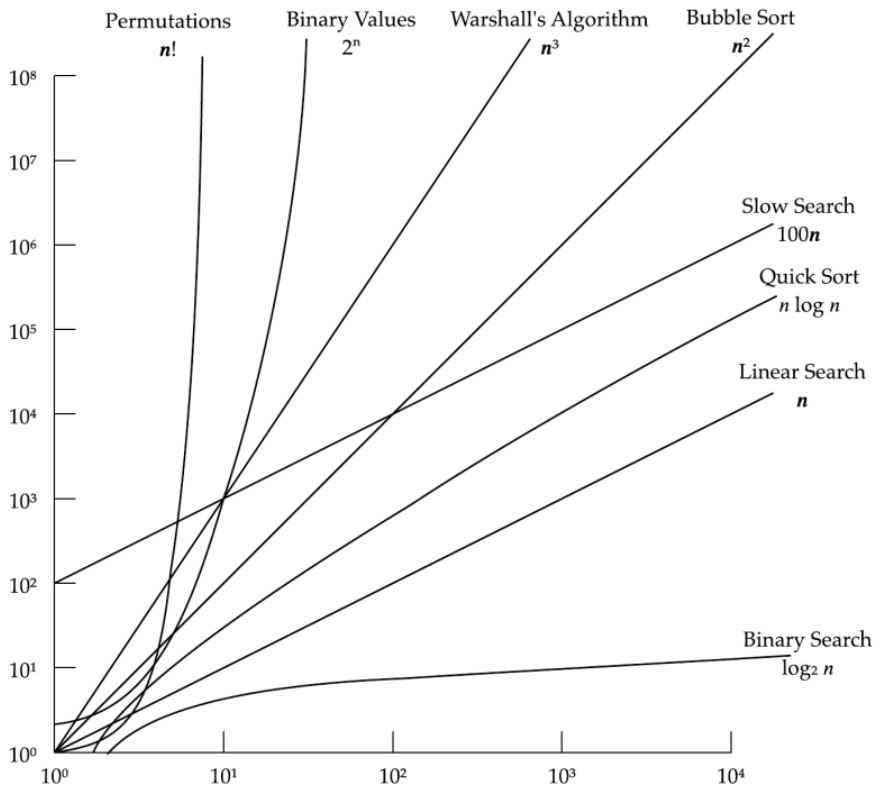

FIGURE 1.1

The growth rates of various functions on a log-log scale: each vertical and horizontal division represents growth by a factor of ten. (Warshall's Algorithm is explained in 2.5.5.)

We can always make an easy problem hard. Consider Algorithm 1.2. Since its first step has complexity $O(n!)$, the whole algorithm is intractable. Fortunately, we know that tractable $O(n \log n)$ algorithms exist for the same problem. We must therefore distinguish the complexity of a problem from the complexity of an algorithm with which we solve it.

Algorithm 1.2 An intractable way to sort a list.

1. Generate all permutations of the list.
2. Choose a permutation in which its elements are in ascending order.

Conversely, an apparently *hard* problem can prove to be *easy*. Consider the question of finding all anagrams of an English word, i.e., the set of all dictionary words that have the same letters as the given word, but in a different order; e.g., 'who' and 'how' are anagrams. Our first thoughts might be as follows:

Algorithm 1.3 An intractable way to find the anagrams of a word in a dictionary.

1. Generate all permutations of the letters in the given word.
2. Reject those permutations that aren't in the dictionary.

This will work well for short words because their letters have few permutations, but we know that the problem of generating permutations is intractable. For longer words, the following would be better:

Algorithm 1.4 A tractable way to find the anagrams of a word in a dictionary.

1. Consider each word in the dictionary.
2. Check if it is an anagram of the given word.

We can check if one word is an anagram of another in time proportional to the length of the word. (One way is to count the number of occurrences of each letter.) Since the dictionary is of fixed length, the total time taken is only proportional to the length of the given word.

How can we know the *inherent complexity of a problem*, that is to say, the complexity of the best possible algorithm that can solve it? In some cases, the answer is easy. In the case of generating permutations, the length of the output sets a lower limit to the complexity of the problem. All we have to assume is that it takes n times longer to output n permutations than it does to output one permutation. On the other hand, if we already *know* a polynomial-time algorithm that solves the problem, then the problem itself must be tractable.

Some problems exist in a kind of limbo. Nobody can prove that the problems are intractable, but nobody has been able to find a polynomial-time algorithm that solves any of them. Some of these problems may actually be intractable; some may not.

An important sub-class of these problems are called **NP-complete**. They have the following properties:

- No polynomial-time algorithm has been discovered for any of them.
- Given the *solution* of an NP-complete problem, we can prove it is correct in polynomial time.
- If anyone ever finds a polynomial-time algorithm to solve any of them, then *all* the NP-complete problems have polynomial-time algorithms.

NP-complete problems have the nature of puzzles: the answers are hard to find, but once found, they are obvious. NP-completeness is the basis of cryptography. One popular method of encrypting (the RSA algorithm) is based on numbers that are products of large primes. Anyone can take two large primes and multiply them, but it is by no means easy for someone else to factorise the resulting number.

It is important to know that this class of problem exists:

I am reminded of one of the first programming assignments I was ever given. It was to find the best sequence in which to insert disks into a drive, given that several programs wanted to share use of the disks, but not necessarily in the same order. For example, one program might need to use disks *A*, *B*, and *C*, in that order, another might need to use *B*, *D*, then *C*, and a third might need to use *C*, *D*, *E*, then *B*. (In this case, one optimal way to mount the disks is the sequence [*A, B, C, D, C, E, B*].) I devised several algorithms to deal with this problem, but each had its weakness: I could always devise a case for which it didn't find the shortest sequence. Eventually I came to the conclusion that the only way I could be sure of finding the shortest solution was to try all possible sequences of length 1, then length 2, and so on, until one was found that solved the problem. Sadly, such an algorithm would be intractable, so I grudgingly settled for a method that got good answers most of the time.

That was a long time ago, when I had never heard of NP-complete problems.[12] I know now that the problem was NP-complete, so I don't feel grudging any more. Indeed, if I *had* solved it, a lot of mathematicians and computer scientists would be out of work.

Remember this story the next time you spend weeks solving a problem, only to find that the answer was obvious with 20/20 hindsight![13] Perhaps it was NP-complete.

Almost all optimisation problems are intractable. For example, scheduling sets of jobs in a machine shop to achieve delivery deadlines or to maximise the use of machinery are both intractable problems, as is creating the shortest examination timetable for a university.

How do I know that *my* problem was NP-complete? Because it is virtually the same as a known NP-complete problem, called *Shortest Super-sequence*.

In practice, that is how any problem is shown to be NP-complete. We try to find a way of mapping a known NP-complete problem, *K*, onto our own problem.[14] If we can do that, then if we can find a way to solve our own problem in polynomial time, then there would also be a way to solve *K* (by mapping it onto our own problem), and therefore *all* NP-complete problems, in polynomial time — and we shouldn't be prepared to believe that!

Such a step proves that our problem is NP-hard, at least as hard as any NP-complete problem, although it might be harder. To show that it is NP-complete, we should show that our problem can also be mapped onto a known NP-complete problem.

---

[12] So long ago, in fact, that neither had anyone else.

[13] Or 6/6 in metric units.

[14] There are books that list known NP-complete problems. See 1.8.

Why are such problems called NP? 'NP' is an abbreviation for 'non-deterministic polynomial'. If we had a computer that, when faced with a decision, could guess (or non-deterministically decide) the right choice to make, then it would need only polynomial time to find and verify a solution. Unfortunately, given the deterministic nature of actual computers, they must explore alternatives systematically, and may take exponential time to hit on the right answer.[15]

On average, we improve things by using *heuristics*. A heuristic is a rule for making correct guesses most of the time. This can lead to two kinds of algorithms: methods that always take polynomial time, but don't guarantee to get an optimal result; and methods that always get an optimal result, but don't guarantee to take polynomial time. It is characteristic of heuristic methods that, given the heuristic, we can always devise problems that defeat the heuristic. The heuristic is useful only if such cases are rare in practice.

How does tractability affect the problems of systems analysis and synthesis?

First, in the worst case, not only don't we know how the system was designed, but neither can we easily prove it is correct. In such a case, it will almost certainly prove incorrect.

Second, although we don't know how the system was designed, it used a means of construction that makes it relatively easy to prove it is correct.[16]

Third, we may have an algorithm for system construction that *guarantees* the system is correct, without the need for proof — as in the example where business rules were converted into a correct procedure. This is the ideal, *provided the construction algorithm is tractable*.

Unluckily for us, many system design problems seem to be intractable. For example, consider Algorithm 1.1, which turns business rules into a computer procedure. We saw that the order in which the conditions are considered affects the quality of the result. A possible means for finding an optimal solution could be that of Algorithm 1.5.

Algorithm 1.5 An intractable way of converting rules into a procedure.

- Consider every possible ordering of the conditions,
- Generate the corresponding procedure, taking the conditions in the order given,
- Choose the best solution.

Since the first of these steps is intractable, the whole method is intractable and cannot be recommended except when the number of conditions is small. It would be intellectually dishonest to tell you to use such an intractable method. So we shall

[15] This is one reason people are excited about the idea of quantum computers; in principle, they are non-deterministic.

[16] This was the basis of an argument used to promote the use of structured programming, contrasted with the unbridled use of 'go to'.

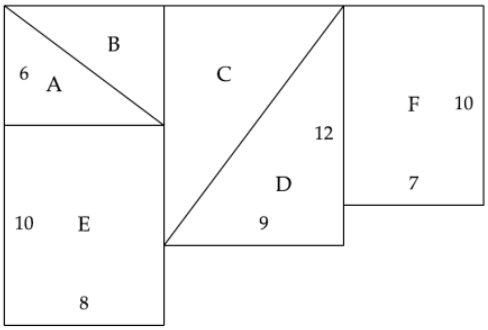

FIGURE 1.2

A puzzle using paper shapes. Cut the pieces from a sheet of paper (not this one!), and reassemble them as explained in the text. Dimensions are in centimeters.

sometimes be forced to recommend heuristic methods. For example, in this problem, it would *usually* be best to deal first with those conditions that appear in the greatest number of rules. Such a heuristic guarantees a quick solution, and usually a good solution, but it cannot promise to always find the best solution.

1.6 MENTAL BLOCKS

Try a little experiment, which I once saw demonstrated by Edward de Bono, of *Lateral Thinking* fame. All you need are a sheet of paper, a ruler, a pair of scissors and a pen or pencil.

Cut out a rectangle 6 cm by 8 cm from the paper, and, in landscape mode, cut it diagonally from bottom right to top left to make two equal triangles. Label these pieces *A* and *B*. Now cut out a second rectangle 9 cm by 12 cm, and again in landscape mode, cut it diagonally from bottom right to top left. Label these pieces *C* and *D*. Now cut a rectangle 10 cm by 8 cm, and label it *E*. Finally, cut a rectangle 7 cm by 10 cm, and label it *F*. (See Figure 1.2.)

The pieces make a puzzle. Assemble the pieces to make the neatest shape you can, and if you need a definition of 'neatest', you can assume it means 'having the shortest perimeter'. The pieces are to be laid flat, on a flat surface, lettered side up, without overlapping. There are no tricks in the way the problem is worded; the pieces fit together like a jigsaw puzzle or a tangram.

Start by assembling pieces *A* and *B*. Now add piece *E*. Next, add pieces *C* and *D*. At this point, most people will have assembled a 16 cm by 17 cm or 16 cm by 20 cm rectangle with a corner missing. Unfortunately, piece *F* doesn't supply the missing corner. See if you can improve your solution, but don't waste too much time, because you can easily find the best solution in a different way.

What has gone wrong? This puzzle is an example of an NP-hard problem, and one can only find the neatest shape by trying every possible arrangement, or by trusting in a heuristic. The six pieces have a total of 20 edges, and each move involves pairing two edges. The first move involves over 150 choices, and there are five moves in all: too many choices to consider. Consciously or unconsciously, most people, given the goal of finding the neatest shape, use a heuristic, and assemble the neatest shape at each stage of the assembly.

Now start over from the beginning. Assemble pieces *E* and *F*. Now add piece *C*. Next, add piece *A*. Finally, add pieces *B* and *D*. If all has gone well, using the same heuristic, you will have assembled a rectangle that is nearly square. If you already got this solution without help, go the head of the class! If you didn't get it *with* help, check the instructions carefully, get it right, and then try the experiment on a friend.[17]

What is the message behind this experiment? The point is that, given the pieces in the first order, even after they are all present, the best solution remains elusive. In the second order, the solution should be obvious. In short, if you try to solve a problem before you have all the pieces, it is a matter of chance whether you will find the solution, and you may block yourself from ever finding it. This is a real phenomenon, and it results directly from how our brains work. The moral is, 'Don't make a theory to fit the facts until you have all the facts, otherwise you will find yourself fitting the facts to the theory'.[18]

The trouble with such advice is that most of us can only think about systems in the ways we are used to and can only absorb facts if we fit them into our evolving design. One problem this book will try to solve is to let you form a working solution without it blocking you from finding the ideal solution. As a means to this end, we need to present ways to *specify* a system without even thinking about its design, and as a means to *this* end, the next chapter covers some mathematics. With luck, most of it will be familiar to you.

This use of specifications leads to another unconventional characteristic of this book. Many books deal with the various stages of system design in the order they are used in practice — perhaps beginning with the process of interviewing the client — but this book is goal-directed. We begin by looking at the *output* of the interviewing process: a relatively formal description of the system, an evolving model. The act of building up this formal model then directs the interviewing process, which we describe later.

---

[17] I used this puzzle in university classes for several years. The pieces of paper were placed on an overhead projector and their shadows were projected for the whole class to see. The class elected a 'volunteer' to assemble the pieces. He (it always seemed to be a male) never found the correct rectangle, and the rest of the class were then eager to improve his solution and to shuffle the pieces. No one ever found the solution this way. Then, out of a kind heart, I would give the pieces to the original volunteer, but in the second order, and he would find the solution right away.

[18] This is the essence of detective fiction. The author presents you with facts in an order that will make you form one wrong theory after another, while the fictional detective pontificates with some pronouncement like, 'Me, I never make theories, I regard only the facts'.

1.7 SUMMARY

In his 1969 book, 'The Mechanism of the Mind' (ISBN 0-14-013787-4), Edward De Bono gives this analogy of the mind[19]:

> Think of a landscape with streams, rivers, and lakes. When rain falls, it runs along the streams and rivers, making them deeper, flooding the lakes. Think of the rain as stimuli or problems, the streams as thoughts, and the lakes as solutions. New information follows well-worn pathways, and new pathways rarely form.
>
> Think of the brain as the land. Although the land directs the rain, it is the rain that has carved the land. The land has been passive. The brain is merely the *passive* result of its experiences.

This rather depressing analogy suggests that we are totally devoid of creativity. But there is hope: the hope is education, which can dig new channels to direct our thoughts. Sadly, much education is implementation-directed. The building engineer learns how to use an I-section girder; the computer programmer learns to use a particular software development kit or learns what sorts of things a binary search tree is useful for. Thus, we learn to think in terms of potential solutions, the lakes where the rain finally comes to rest. Problems flow by the steepest path directly to the nearest solution. In this book, I hope to dig some channels that lead to reservoirs high in the hills, from which we can direct our thoughts towards the most appropriate solutions.

In his 1967 book, 'The Use of Lateral Thinking' (ISBN 0-14-013788-2), De Bono does some civil engineering, showing us how to build dams and divert flows: we should always question our first thoughts, and maybe our second thoughts too. His book suggests several other tricks that we can use. One, for example, is to pick a word from the dictionary with a pin, and ask how that helps our problem. The brain is passive. We must seek inspiration from outside. I hope to offer you inspiration in this book, by getting you to think about familiar problems in unfamiliar ways.

There is nothing new in this book. It is a kit of parts I have assembled into a reasonably coherent system. This has meant drawing topics from many sources, although none of them very deeply. This seems to be the first time all these topics have all been brought together in one place. I can take no credit for this. It is merely the passive result of my experiences.

One thread that runs through the book is the question of tractability. I have avoided, as far as possible, suggesting that you solve intractable problems. For example, many books on database design will include an instruction such as, 'Find a minimal cover G of the set of FDs F', as if it were no trouble at all.[20] Although an

[19] This is my paraphrase, not a quotation.

[20] A cover of F is a set of FDs with the same closure as F. Finding the closure of a set of FDs is an exponentially hard problem. Fortunately, there is an efficient shortcut for finding a minimal cover, which we describe in 4.4.3.

experienced designer can often find such a cover intuitively, experience may be the one thing you don't have.

1.8 FURTHER READING

The complexity of algorithms is discussed much more comprehensively in the first three chapters of 'Foundations of Computer Science: C Edition' by Alfred V. Aho and Jeffrey D. Ullman (1994, ISBN: 0-7167-8284-7). Although this excellent book is out of print, PDFs of the book are available on the Internet.

'Computers and Intractability: A Guide to the Theory of NP-Completeness' (1979, ISBN: 0-7167-1045-5), is a classic textbook by Michael Garey and David S. Johnson, which introduces the reader to complexity theory and lists over 300 distinct NP-hard problems. It is a useful guide to any computer scientist who wants to know if their own particular problem is intractable.

The RSA encryption algorithm was invented by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT and is named after the initial letters of their surnames. It was published in 1978 in *Communications of the ACM* 21 (2): 120–126 as 'A Method for Obtaining Digital Signatures and Public-Key Cryptosystems'. It was the first practical public-key encryption system. Later, it came to light that essentially the same idea had been invented a few months earlier by Clifford Cocks at the Government Communications Headquarters in the UK, but his work was classified and had to remain secret for many years.

Quicksort is one of several sorting algorithms with average complexity $O(n \log n)$ — although its worst-case performance is $O(n^2)$. It was published in 1961 by Tony Hoare as 'Algorithm 64: Quicksort', *Communications of the ACM* 4 (7): 321.

Edsger W. Dijkstra was an early advocate of constructing programs that are provably correct. His 1976 book, 'A Discipline of Programming' (ISBN: 0-13-215871-X), explains how to derive procedures from their pre-conditions and post-conditions. He also showed how to use 'loop invariants' to prove the correctness of procedures containing loops — although the choice of a suitable invariant is sometimes tricky. This book was also influential in promoting the use of structured programming.

1.9 EXERCISES

1. Apply Algorithm 1.1 to the set of rules on page 10 again, this time picking the other condition first. Verify that it produces the same result as the 'intuitive' solution.
2. Suppose you have to sort a list of length l containing integers in the range $1 - n$. Is it possible to sort them in time with complexity less than $O(l \log l)$?
3. A computer has a word length of n bits. Is the problem of listing each possible word tractable or intractable?

CHAPTER

Mathematical background

2

CHAPTER CONTENTS

Systems Analysis and Synthesis: Bridging Computer Science and Information Technology. DOI: 10.1016/B978-0-12-805304-1.00011-4

INTRODUCTION

There are five closely related mathematical concepts that unify the following chapters: propositional calculus, predicate calculus, set theory, the algebra of relations, and graph theory. We won't need to study any of them in depth, but by referring to these concepts, we shall save a lot of long-winded explanation later on. Many readers will already be familiar with these topics, and they will merely need to check how the notation in this book compares with what they are already familiar with. Indeed, if you know what is meant by the strongly connected components of a graph, you already understand a key concept in what follows.

For those readers unfamiliar with the material, I shall explain it in three ways: informally, algebraically, and pictorially. Later on, here and there, I shall venture into some other branches of mathematics, but they will be explained as we go along. I present only notation and a few key theorems here, no formal proofs. This chapter is a *summary*, not a tutorial. Since the material in this chapter could fill a textbook, the reader who needs a more tutorial style is advised to study one or more of the textbooks listed in Section 2.9.

If you find the chapter heavy going, you may prefer to skim it now and come back to it when the text refers to it.

Our aim in this chapter is to understand some key concepts in graph theory, such as *closure*, *reduction*, and *transitive root*. *Graphs* are used to visualise relations between sets. *Sets* and *relations* are often defined by *predicates*, and predicate calculus is easily understood following a discussion of *propositions*.

2.1 PROPOSITIONAL CALCULUS

Propositional calculus is the mathematics of Boolean variables, so the basic ideas should be familiar to most readers. We shall not make much *direct* use of it in this book, but it will help to define several ideas presented later in this chapter.

A **proposition** is a statement that is either true or false, such as, 'Peter is male', 'John is female', or 'Peter has parent Mary'.

2.1.1 LOGICAL OPERATORS

Propositions can be combined using **logical operators** to form **logical expressions**. The most frequently used operators are as follows (where the literals P and Q are arbitrary propositions):

$\neg P$	'**not** P'.
$P \wedge Q$	'P **and** Q'.
$P \vee Q$	'P **or** Q'.
$P \Rightarrow Q$	'P **implies** Q'.
$P \Leftarrow Q$	'P **if** Q', or 'P **is implied by** Q'.
$P \Leftrightarrow Q$	'P **if and only if** Q', or 'P **is equivalent to** Q'.

The results of the operators are defined by the following *truth table*,

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftarrow Q$	$P \Leftrightarrow Q$
False	*False*	*True*	*False*	*False*	*True*	*True*	*True*
False	*True*	*True*	*False*	*True*	*True*	*False*	*False*
True	*False*	*False*	*False*	*True*	*False*	*True*	*False*
True	*True*	*False*	*True*	*True*	*True*	*True*	*True*

Given two component expressions P and Q, a **truth table** shows the value of each expression for each combination of values of P and Q. *True* and *False* are constants. The operators are displayed from left to right in order of precedence, with 'not' ($\neg$) binding the tightest. That is to say, the expression $P \wedge Q \vee \neg P \wedge R \Rightarrow Q \vee R$ means the same as $(P \wedge Q) \vee ((\neg P) \wedge R) \Rightarrow (Q \vee R)$.

$\neg P$ is called the **logical complement** of P.[1]

[1] An electronics engineer would write $\bar{P}$.

The $\vee$ (or) operator has the meaning often written as 'and/or'. Its everyday meaning does not always match its logical meaning. This can confuse the uninitiated.[2]

Beginners often find the entries for $P \Rightarrow Q$ confusing, especially because $False \Rightarrow True$ has the value *True*. Consider the case 'Pluto is a dog' $\Rightarrow$ 'Pluto is an animal'. The table does not say that if 'Pluto is a dog' is *False* then 'Pluto is an animal' *must* be *True*, it merely says that there is nothing wrong with a universe in which Pluto is an animal because Pluto is a cat. Likewise, in the case when P and Q are both *False*, there is nothing wrong with a universe where Pluto is not an animal because Pluto is a minor planet. The main points to remember are that if $P \Rightarrow Q$, when P is *True*, Q must be *True*, and when Q is *False*, P must be *False*, but knowing that P is *False* tells us nothing about Q, and knowing that Q is *True* tells us nothing about P.

The word 'implies' can itself be a source of confusion. $P \Rightarrow Q$ does not mean 'P causes Q', but is best read as 'P is a special case of Q', as in, 'A dog is a special case of an animal'.

Equivalently, the operators may be defined by using the following axioms:

$$True = \neg False \tag{2.1.1}$$
$$False = \neg True \tag{2.1.2}$$
$$P \wedge True = P \tag{2.1.3}$$
$$P \wedge False = False \tag{2.1.4}$$
$$P \vee True = True \tag{2.1.5}$$
$$P \vee False = P \tag{2.1.6}$$
$$P \Rightarrow Q = \neg P \vee Q \tag{2.1.7}$$
$$P \Leftarrow Q = Q \Rightarrow P = P \vee \neg Q \tag{2.1.8}$$
$$P \Leftrightarrow Q = (P \wedge Q) \vee (\neg P \wedge \neg Q) \tag{2.1.9}$$

Axioms 2.1.1 and 2.1.2 define negation, Axioms 2.1.3 and 2.1.4 define the $\wedge$ operator, and Axioms 2.1.5 and 2.1.6 define the $\vee$ operator. Axioms 2.1.7, 2.1.8, and 2.1.9 define the implication operators in terms of more basic operators. From these axioms, we can easily prove these two well-known laws:

$$P \wedge \neg P = False, \qquad P \vee \neg P = True \tag{2.1.10}$$
$$(P \Rightarrow Q) \wedge (Q \Rightarrow R) \Rightarrow (P \Rightarrow R) \tag{2.1.11}$$

Theorem 2.1.10 is called the **law of the excluded middle**, and Theorem 2.1.11 demonstrates the transitivity of the $\Rightarrow$ operator.

---

[2] In an an old story, an army general is confronted by a computer intended to advise him on military strategy. The dialogue goes like this:
General: Shall I advance or retreat?
Computer: `Yes.`
General: Yes what?
Computer: `Yes, sir!`

2.1.2 PROPERTIES OF LOGICAL OPERATORS

We can always prove whether two propositional expressions are equivalent by writing out their truth tables. Unfortunately, this approach is inherently *intractable*, because an expression containing N propositions requires a table containing 2^N entries. It is often more practical to work algebraically. Some useful theorems are listed below.

$$P \wedge Q = Q \wedge P \tag{2.1.12}$$
$$P \vee Q = Q \vee P \tag{2.1.13}$$
$$(P \wedge Q) \wedge R = P \wedge (Q \wedge R) = P \wedge Q \wedge R \tag{2.1.14}$$
$$(P \vee Q) \vee R = P \vee (Q \vee R) = P \vee Q \vee R \tag{2.1.15}$$
$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R) \tag{2.1.16}$$
$$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R) \tag{2.1.17}$$
$$P \wedge P = P \tag{2.1.18}$$
$$P \vee P = P \tag{2.1.19}$$
$$\neg(\neg P) = P \tag{2.1.20}$$
$$\neg(P \wedge Q) = \neg P \vee \neg Q \tag{2.1.21}$$
$$\neg(P \vee Q) = \neg P \wedge \neg Q \tag{2.1.22}$$

Theorems 2.1.12 and 2.1.13 mean that the operands of $\wedge$ and $\vee$ *commute*, i.e., may appear in either order. Theorems 2.1.14 and 2.1.15 mean that $\wedge$ and $\vee$ are *associative*, i.e., that a string of $\wedge$ or $\vee$ operators may be evaluated in any order (and therefore the order needn't be spelled out). The distribution laws of 2.1.16 and 2.1.17 allow products to be expanded into simple terms. The idempotence laws of 2.1.18 and 2.1.19 allow multiple occurrences of the same variable to be simplified. Equation 2.1.20 expresses the idea that two negatives make a positive. Finally, **De Morgan's laws**, shown in 2.1.21 and 2.1.22, allow '$\wedge$' to be replaced by '$\vee$', or *vice versa*. A generalisation, known as **De Morgan's Theorem**, states that the logical complement of any proposition can be found by replacing '$\wedge$' by '$\vee$' and '$\vee$' by '$\wedge$' and complementing each basic proposition.

Informally, we often see $P \wedge Q$ written as $P.Q$, $P \vee Q$ written as $P + Q$, and $\neg P$ as $-P$. This is because, with such an interpretation, logical expressions obey some of the rules of ordinary school algebra — except that $P + P = P$, etc.[3] Thus, we often see $P \wedge Q$ referred to as a 'product', and $P \vee Q$ referred to as a 'sum'.

2.1.3 CONJUNCTIVE NORMAL FORM

To prove that two expressions are equal, a frequently used technique is to transform both expressions to a standard form. One such standard form is called **conjunctive normal form** or **CNF**. An expression in CNF is a 'product of sums'. The 'sums' are

[3] School algebra correctly models the distribution law of equation 2.1.16, but fails to express that of equation 2.1.17.

literals (simple propositions or negated propositions, e.g., P, or $\neg Q$) linked by $\vee$, which are then formed into a 'product' using $\wedge$.[4]

Consider the expression

$$(P \Leftrightarrow Q) \tag{2.1.23}$$

Its conjunctive normal form is

$$(\neg P \vee Q) \wedge (P \vee \neg Q) \tag{2.1.24}$$

To get this result, (using Axiom 2.1.9) we reduce all the operators to $\wedge$, $\vee$, and $\neg$

$$(P \wedge Q) \vee (\neg P \wedge \neg Q) \tag{2.1.25}$$

We then use the first distribution law, three times:

$$\begin{aligned}(P \wedge Q) \vee (\neg P \wedge \neg Q) &= (P \vee \neg P \wedge \neg Q) \wedge (Q \vee \neg P \wedge \neg Q) \\ &= (P \vee \neg P) \wedge (P \vee \neg Q) \wedge (Q \vee \neg P \wedge \neg Q) \\ &= (P \vee \neg P) \wedge (P \vee \neg Q) \wedge (Q \vee \neg P) \wedge (Q \vee \neg Q)\end{aligned}$$

Finally, we eliminate the sums $(P \vee \neg P)$ and $(Q \vee \neg Q)$, which are always *True*, leaving

$$(P \vee \neg Q) \wedge (Q \vee \neg P) \tag{2.1.26}$$

Normalisation is a purely mechanical process that a computer can do (although it is NP-hard).

We can prove the theorem

$$(P \Leftrightarrow Q) = (P \Rightarrow Q) \wedge (P \Leftarrow Q) \tag{2.1.27}$$

by converting both sides to CNF. We have already dealt with the left-hand side above. Normalising its right-hand side is left as a simple exercise for the reader.[5]

---

[4] An interesting connection exists between CNF and the notion of NP-completeness discussed in the previous chapter. Consider the question of whether any assignment of values exists that makes a CNF expression *False*. All we need to do is render any one sum *False* and the whole conjunction will be *False*. To do this, we merely need to make each literal in the sum *False*. For example, in the CNF, $(\neg P \vee Q) \wedge (P \vee \neg Q)$, we can either set P = *True* and Q = *False*, rendering the first sum *False*, or set P = *False* and Q = *True*, rendering the second sum *False*.

Sadly, the question of finding a set of values that makes a given CNF expression *True* isn't so easy, because we have to find a set of assignments that make *all* the sums *True* simultaneously. In fact (for three literals or more), this was the first problem to be considered to be NP-complete.

The astute reader will realise that, equivalently, we could find the converse (negation) of the expression using De Morgan's Theorem, and find an assignment for which it is *False*. But, needless to say, expressing the converse of the expression in CNF can itself can take time 2^N.

[5] There are several normal forms for Boolean expressions. For example, **DNF** (**Disjunctive Normal Form**) is expressed as a sum of products.

2.1.4 LOGICAL INFERENCE

In a **logical proof**, it is conventional for propositions to be stacked in columns. Stacked propositions are implicitly linked by $\wedge$. From the propositions already known, new propositions can be deduced using **rules of inference**. These are usually stacked below those already known as the proof progresses, with the final conclusion separated from the others by a horizontal line.

The best-known rule is called **modus ponens**: if P implies Q and P is true, then Q must be true.

$$\frac{\begin{array}{c} P \Rightarrow Q \\ P \end{array}}{Q}$$

Its converse is called **modus tollens**: if P implies Q and Q is false, then P must be false.

$$\frac{\begin{array}{c} P \Rightarrow Q \\ \neg Q \end{array}}{\neg P}$$

Some rules are trivial, such as **and elimination**, and **or introduction**:

$$\frac{P \wedge Q}{P}$$

$$\frac{P}{P \vee Q}$$

All these rules are special cases of one general rule, called **resolution**:

$$\frac{\begin{array}{c} P \vee Q \\ \neg P \vee R \end{array}}{Q \vee R}$$

P must be either *True* or *False*, so either Q or R must be *True*. P is said to **cancel**, leaving $Q \vee R$ as the **resolvent**. Resolution is well-suited to automated proof, because it is merely necessary to find two propositions containing complementary terms. This is easiest if propositions are simple literals linked by $\vee$. Thus logical proof is straightforward if propositions are in CNF.

2.2 FIRST-ORDER PREDICATE CALCULUS

The (first-order) **predicate calculus** is the mathematics of Boolean functions, so, again, most readers should find the basic ideas familiar.[6]

[6] In *first-order* predicate calculus, the predicates themselves cannot be variables.

A **predicate** is a sentence that may be *True* or *False*, depending on the values of the variables it contains. For example, 'm is male', 'f is female', and 'c has parent p' are predicates whose truths depend on m, f, c, and p. Predicates are usually written the same way as Boolean *functions* are written, e.g., $Male(m)$, $Female(f)$, or $Has\ Parent(c,p)$, with the convention that constants are capitalised, and variables are written in lower case.

2.2.1 QUANTIFIERS

First-order predicate calculus uses the same set of operators as propositional calculus, but it also employs **quantifiers**. We use an 'A' upside-down ($\forall$) to mean 'for all', and an 'E' backwards ($\exists$) to mean 'there exists'. For example, to represent the usual meaning of the sentence, 'Everybody has a parent', we would write

$$\forall c(\exists p(Parent(c,p))) \tag{2.2.1}$$

unless we thought it really meant, 'There is somebody who is everybody's parent', which would be written

$$\exists p(\forall c(Parent(c,p))) \tag{2.2.2}$$

Provided we are clear about the way we interpret predicates, predicate calculus is less ambiguous than natural language and is therefore widely used to specify computer programs.

Variables preceded by $\forall$ are said to be **universally quantified**. Variables preceded by $\exists$ are said to be **existentially quantified**.

If x can take values x_1, x_2, and x_3, then $\forall x P(x)$ means $P(x_1) \wedge P(x_2) \wedge P(x_3)$, and $\exists x P(x)$ means $P(x_1) \vee P(x_2) \vee P(x_3)$. In other words, $\forall$ functions like $\wedge$, and $\exists$ functions like $\vee$. As a result, the following extensions of De Morgan's laws are sometimes useful.[7]

$$\neg(\forall x P(x)) \Leftrightarrow \exists x(\neg P(x))$$
$$\neg(\exists x P(x)) \Leftrightarrow \forall x(\neg P(x))$$

Because the names we choose for constants have no inherent meaning to predicate calculus, we must always reason formally. For example, from *Has Parent*(*Peter*, *Mary*), we cannot deduce *Has Child*(*Mary*, *Peter*). Although *Has Parent*(*Peter*, *Mary*) = *True* suggests to *us* that *Has Child*(*Mary*, *Peter*) = *True*, predicate calculus cannot deduce it, unless it has some additional rule with which to reason, such as $Has\ Child(p,c) \Leftrightarrow Has\ Parent(c,p)$.

As long as we deal with *finite* sets of variables, predicate calculus remains **complete**: in principle, any theorem can be proved by enumerating all possible cases. On the other hand, once we deal with infinite sets, such as the integers, predicate calculus becomes capable of expressing the deepest problems in mathematics, including unresolvable paradoxes.

[7] Since P here is a variable that stands for *any* unary predicate, these theorems are not first-order, they are 'higher-order'.

2.3 SETS

A **set** is any collection of objects, called its **elements**. Normally the elements have the same type (e.g., *integers* or *animals*), in which case the set is said to be **typed**. The set of all elements of that type is called the **universe of discourse**, often symbolised by **U**. (Sets of elements of mixed types are often referred to as **collections**.)

2.3.1 SET NOTATION

A set of three males may be written as $\{Peter, Paul, Mark\}$. The members of a set have no order, so $\{Paul, Mark, Peter\}$ is the same set. Nor does it matter if elements are duplicated, so $\{Peter, Paul, Mark, Peter\}$ is the same set too. Sets may be given names, such as $Males = \{Peter, Paul, Mark\}$.

A special case is an empty set (written as $\{\}$ or $\emptyset$), which has no elements. In a sense, all empty sets are the same, but if we are using **typed sets**, then the empty set of males would be considered distinct from the empty set of females.

We write $T \subseteq S$ to say that T is a **subset** of S. A subset of S is any set whose elements are all elements of S. The set *Males* has 8 subsets:

$$\{Peter, Paul, Mark\},$$
$$\{Paul, Mark\},$$
$$\{Peter, Mark\},$$
$$\{Peter, Paul\},$$
$$\{Peter\},$$
$$\{Paul\},$$
$$\{Mark\},$$
$$\{\}$$

Note that *Peter* isn't a *subset* of $\{Peter, Paul, Mark\}$; *Peter* is an *element*, not a set.

If we want to say that element x is a member of set S, we write $x \in S$. To say that x is *not* a member of S, we write $x \notin S$.

The number of elements of set S is called its **cardinality**, $|S|$. A set with cardinality $|S|$ has $2^{|S|}$ subsets. For this reason, the set of all subsets of S, called the **powerset** of S, may be written as 2^S.

Any set is a subset of itself. If we exclude this possibility, we refer to the subset as a **proper subset** and write $T \subset S$.

The elements of a set don't have to be simple. They may, for example, consist of pairs of values, such as the set

$$Has\ Parent = \{(Peter, Mary), (Peter, Mark), (Paul, Mary), (Paul, Mark)\}$$

Sets don't have to be finite, or even countable; we may speak of the set of integers, or the set of real numbers, although in such cases we cannot write down the set as a

list of elements. Usually, we define such sets recursively, e.g.,

$$\begin{cases} 0 \in \mathit{Integers}, \\ \forall n(n \in \mathit{Integers} \Rightarrow (n+1) \in \mathit{Integers}), \\ \forall n(n \in \mathit{Integers} \Rightarrow (n-1) \in \mathit{Integers}). \end{cases} \tag{2.3.1}$$

We often define sets using the notation,

$$S = \{E(x, y, \ldots) \mid P(x, y, \ldots)\} \tag{2.3.2}$$

where S is the set being defined, $E(x, y, \ldots)$ is an expression in one or more variables, and $P(x, y, \ldots)$ is a predicate. (Out loud, we say, 'The set of all $E(x, y, \ldots)$ such that $P(x, y, \ldots)$.') For example, we can write,

$$\mathit{Squares} = \{n^2 \mid n \in \mathit{Integers}\} \tag{2.3.3}$$

to define the set of the squares of the integers. The expression is read as, '*Squares* is the set of all n^2 such that n is an integer'.

2.3.2 SET OPERATORS

The **union** ($\cup$) of sets S and T is the set of all elements that are members of *either* S *or* T. We write

$$S \cup T = \{x \mid x \in S \vee x \in T\} \tag{2.3.4}$$

which we read as, 'The union of S and T is the set of all elements x such that either x is a member of S or x is a member of T'.

If $\mathit{Males} = \{\mathit{Peter}, \mathit{Paul}, \mathit{Mark}\}$ and $\mathit{Females} = \{\mathit{Mary}, \mathit{Jane}\}$, then

$$\begin{aligned} \mathit{Persons} &= \mathit{Males} \cup \mathit{Females} \\ &= \{\mathit{Peter}, \mathit{Paul}, \mathit{Mark}, \mathit{Mary}, \mathit{Jane}\} \end{aligned}$$

Similarly, if $\mathit{Children} = \{\mathit{Peter}, \mathit{Mark}, \mathit{Mary}\}$, and $\mathit{Parents} = \{\mathit{Paul}, \mathit{Jane}, \mathit{Mark}, \mathit{Mary}\}$, then $\mathit{Children} \cup \mathit{Parents} = \{\mathit{Peter}, \mathit{Paul}, \mathit{Mark}, \mathit{Mary}, \mathit{Jane}\}$.

It is always the case that $S \subseteq S \cup T$ and $T \subseteq S \cup T$.

The **intersection** ($\cap$) of sets S and T is the set of all elements that are members of both S and T. We write

$$S \cap T = \{x \mid x \in S \wedge x \in T\} \tag{2.3.5}$$

If $\mathit{Children} = \{\mathit{Peter}, \mathit{Mark}, \mathit{Mary}\}$, and $\mathit{Parents} = \{\mathit{Paul}, \mathit{Jane}, \mathit{Mark}, \mathit{Mary}\}$, then $\mathit{Children} \cap \mathit{Parents} = \{\mathit{Mark}, \mathit{Mary}\}$. Not surprisingly, $\mathit{Males} \cap \mathit{Females} = \{\}$.

It is always the case that $S \cap T \subseteq S$ and $S \cap T \subseteq T$.

The **ordered difference**, or **asymmetric difference**, of S and T consists of all elements of S that are *not* members of T:

$$S \setminus T = \{x \mid x \in S \wedge x \notin T\} \tag{2.3.6}$$

If $Children = \{Peter, Mark, Mary\}$, and $Parents = \{Paul, Jane, Mark, Mary\}$, then $Children \setminus Parents = \{Peter\}$, but $Parents \setminus Children = \{Paul, Jane\}$.

2.3.3 PROPERTIES OF SET OPERATORS

Because the $\cup$ operator is defined using $\vee$, and the $\cap$ operator is defined using $\wedge$, it isn't surprising that set theory contains analogues of the commutation, association, distribution, and idempotence rules of propositional calculus, given in Equations 2.1.12–2.1.19 on pages 27–27.

$$\begin{aligned}
S \cap T &= T \cap S \\
S \cup T &= T \cup S \\
(S \cap T) \cap R &= S \cap (T \cap R) = S \cap T \cap R \\
(S \cup T) \cup R &= S \cup (T \cup R) = S \cup T \cup R \\
S \cap (T \cup R) &= (S \cap T) \cup (S \cap R) \\
S \cup (T \cap R) &= (S \cup T) \cap (S \cup R) \\
S \cup S &= S \\
S \cap S &= S
\end{aligned}$$

To extend the analogy, the empty set $\emptyset$ corresponds to *False*, and the universal set $\mathbf{U}$ corresponds to *True*:

$$\begin{aligned}
S \cap \mathbf{U} &= S \\
S \cap \emptyset &= \emptyset \\
S \cup \mathbf{U} &= \mathbf{U} \\
S \cup \emptyset &= S
\end{aligned}$$

which match 2.1.3–2.1.6 above.

Although set theory has no $\neg$ operator, we can find a similar role for asymmetric difference in Axioms 2.1.1–2.1.2 and Theorem 2.1.10.

(Within the universe of discourse, $\mathbf{U}$, $x \notin S$ if and only if $x \in (\mathbf{U} \setminus S)$.)

$$\begin{aligned}
\mathbf{U} \setminus \mathbf{U} &= \emptyset \\
\mathbf{U} \setminus \emptyset &= \mathbf{U} \\
S \setminus (\mathbf{U} \setminus S) &= S \\
S \cap (\mathbf{U} \setminus S) &= \emptyset \\
S \cup (\mathbf{U} \setminus S) &= \mathbf{U}
\end{aligned}$$

There are even analogues of De Morgan's laws 2.1.21 and 2.1.22:

$$\begin{aligned}
\mathbf{U} \setminus (S \cap T) &= (\mathbf{U} \setminus S) \cup (\mathbf{U} \setminus T) \\
\mathbf{U} \setminus (S \cup T) &= (\mathbf{U} \setminus S) \cap (\mathbf{U} \setminus T)
\end{aligned}$$

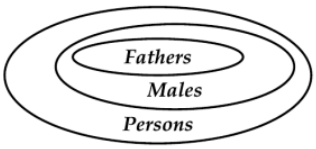

FIGURE 2.1

An Euler diagram showing subset relationships between *Fathers*, *Males*, and *Persons*.

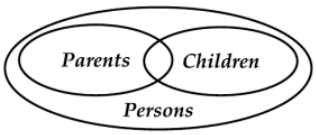

FIGURE 2.2

An Euler diagram showing that *Parents* and *Children* are overlapping subsets of *Persons*.

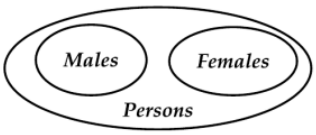

FIGURE 2.3

An Euler diagram showing that *Males* and *Females* are disjoint subsets of *Persons*.

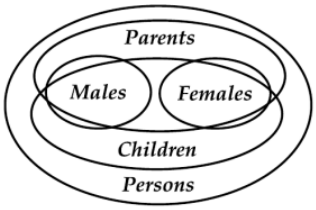

FIGURE 2.4

An Euler diagram that tries to show too much. As a result, it fails to show anything clearly.

2.3.4 EULER DIAGRAMS

We may visualise the relationships between sets by drawing **Euler diagrams**. Figure 2.1 shows that *Males* is a subset of *Persons*; every element of *Males* is also an element of *Persons*. Likewise, every member of *Fathers* is a member of *Males*. The outermost ellipse defines our universe of discourse, the set *Persons*. This gives us a visual way to understand the $\Rightarrow$ operator of predicate calculus: $x \in Males \Rightarrow x \in Persons$, and $x \notin Persons \Rightarrow x \notin Males$. On the other hand, $x \notin Males$ permits either $x \in Females$ (for example) or $x \notin Persons$, and $x \in Persons$ permits either $x \in Males$ or $x \notin Males$. Not surprisingly, like $\Rightarrow$ in Equation 2.1.11, the $\subseteq$ operator is transitive.

Figure 2.2 shows a situation where *Parents* and *Children* **overlap**; at least one element of *Parents* is also an element of *Children*, but not all of them.

Figure 2.3 shows a situation where *Males* and *Females* are **disjoint**; *Males* and *Females* have no common element.

If $S_1, S_2, \ldots S_N$ are all mutually disjoint and $S_1 \cup S_2 \cup \ldots S_N = S$, then $S_1, S_2, \ldots S_N$ are said to **partition** S. For example, if $Persons = Males \cup Females$, and *Males* and *Females* are disjoint, then *Males* and *Females* partition *Persons*.

Often, it is possible to combine several relationships into a single diagram, as in Figure 2.4, which shows that *Males* and *Females* are disjoint sets entirely contained

within the union of the *Parents* and *Children* sets. There is a limit to how much information a single Euler diagram can express: for example, Figure 2.4 doesn't show that the union of *Males* and *Females* is the same as the set of *Persons*.

If we want to show the elements of a set on an Euler diagram, they must be drawn as points, not areas. Elements of a set are not subsets of it — although it is possible to define **singleton** sets, containing only one element, e.g., $\{Peter\}$.

2.4 RELATIONS AND FUNCTIONS

2.4.1 CARTESIAN PRODUCTS

The **Cartesian product** of two sets, X and Y, denoted by $X \times Y$, is the set of *all* ordered pairs (x, y), where x is an element of X and y is an element of Y:[8]

$$X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\} \tag{2.4.1}$$

For example, if $Children = \{Peter, Mark, Mary\}$, and $Parents = \{Paul, Jane, Mark, Mary\}$, then

$$\begin{aligned} Children \times Parents = \{&(Peter, Paul), (Peter, Jane), (Peter, Mark), (Peter, Mary),\\ &(Mark, Paul), (Mark, Jane), (Mark, Mark), (Mark, Mary),\\ &(Mary, Paul), (Mary, Jane), (Mary, Mark), (Mary, Mary)\} \end{aligned}$$

The order of terms within a pair is important: $(Mary, Mark)$ is not the same ordered pair as $(Mark, Mary)$; $X \times Y = Y \times X$ if and only if $X = Y$.

2.4.2 BINARY RELATIONS

The equation $x^2 + y^2 = 1$ describes a *relationship* between x and y. A **binary relation** is a similar notion, but it has a sense of direction, for example, **from** x **to** y.

A **binary relation** R **from domain** X **to codomain** Y is a *selected* set of ordered (x, y) pairs. The first element of each pair is drawn from the domain, and the second element from the codomain. Formally, $R \subseteq X \times Y$, where $X = \text{dom}(R)$ and $Y = \text{codom}(R)$.

As an example, consider

$$Unit\ Circle = \{(x, y) \mid x \in Reals \wedge y \in Reals \wedge x^2 + y^2 = 1\} \tag{2.4.2}$$

This describes a binary relation **from** a real value x **to** (typically) two y values, i.e., $y = +\sqrt{1 - x^2}$ and $y = -\sqrt{1 - x^2}$. For example $(0, +1)$ and $(0, -1)$ are both

[8] The name 'Cartesian' derives from the Cartesian co-ordinate system used in co-ordinate geometry, in which points in a plane are represented as ordered pairs, such as (3.0, 2.5). As in geometry, it is possible to have Cartesian products of higher order, such as $X \times Y \times Z$.

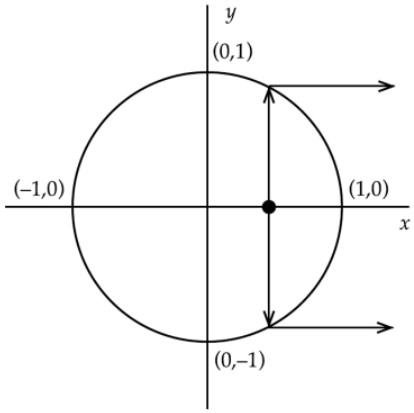

FIGURE 2.5

Unit Circle considered as a relation from x to y.

members of relation *Unit Circle*. (There is only one y value when $x = \pm 1$. See Figure 2.5.)

To emphasise the fact that a relation has direction, we shall write a pair such as (x, y) in the form $x \mapsto y$ (x **'maps to'** y), thus,

$$Unit\ Circle = \{x \mapsto y \mid x \in Reals \land y \in Reals \land x^2 + y^2 = 1\} \tag{2.4.3}$$

An alternative way to express that $x \in X$ and $y \in Y$ is to include the domain and codomain as the type of the relation:

$$Unit\ Circle : Reals \rightarrow Reals = \{x \mapsto y \mid x^2 + y^2 = 1\} \tag{2.4.4}$$

(We shall make particular use of this notation when we discuss schemas.) If $x \mapsto y \in R$, we may alternatively write $x\ R\ y$, for example, $x \mapsto y \in Unit\ Circle$ means the same as $x\ Unit\ Circle\ y$.

As in the case of *Unit Circle*, relations are often referred to by name.

For a second example, please forgive the somewhat circular definition, and consider

$$\leq = \{x \mapsto y \mid x \in Integers \land y \in Integers \land x \leq y\} \tag{2.4.5}$$

In other words, '$\leq$' is the name of the relation that maps integer x to all integers y no less than itself.

When, as here, the domain and codomain of a relation are infinite sets, the mapping cannot be enumerated and must be given by a formula. On the other hand, relations may also be defined over discrete sets, in which case they can be defined by enumeration. For example, consider the relation *Has Parent*, whose domain is {*Peter*, *Mark*, *Mary*} and whose codomain is {*Paul*, *Jane*, *Mark*, *Mary*}, with the

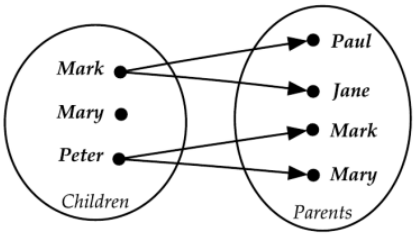

FIGURE 2.6

The *Has Parent* relation *from* the set {*Peter*, *Mark*, *Mary*} *to* the set {*Paul*, *Jane*, *Mark*, *Mary*}.

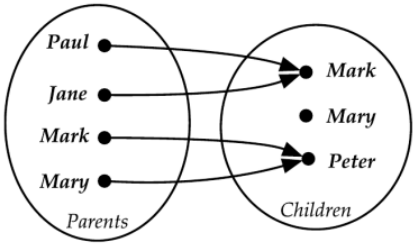

FIGURE 2.7

Has Child, the converse of the *Has Parent* relation.

mapping $\{Peter \mapsto Mary, Peter \mapsto Mark, Mary \mapsto Jane, Mary \mapsto Paul\}$. We would describe this particular relation as being **from** the set $\{Peter, Mark, Mary\}$ **to** the set $\{Paul, Jane, Mark, Mary\}$.

We may visualise finite relations by drawing a mapping, as in Figure 2.6. The ellipses represent the domain and codomain; the black dots, elements; and the arrows, the mapping, i.e., the ordered pairs.

If every element of the domain appears in at least one ordered pair, the relation is **total**; otherwise it is **partial**. If every element of the codomain appears in at least one pair, the relation is **onto** its codomain; otherwise it is **into** its codomain. The relation in Figure 2.6 is partial and onto.

Every relation has a **converse**, which is obtained by exchanging the roles of the domain and codomain and reversing the mapping.

$$R^{-1} = \{y \mapsto x \mid x\ R\ y\} \tag{2.4.6}$$

The converse of the *Has Parent* relation, which we may call *Has Child*, is illustrated in Figure 2.7. In general, it is a relation, because parents can have any number of children, including zero, but in this particular example, each parent has exactly one child.

It is possible to have relations of orders higher than binary. A ternary relation is defined by

$$R \subseteq X \times Y \times Z \tag{2.4.7}$$

For our purposes, such relations will always be reduced to binary relations. We might regard $X \times Y$ as the domain and Z as the codomain, i.e.,

$$R : X \times Y \to Z = \{(x, y) \mapsto z \mid P(x, y, z)\} \tag{2.4.8}$$

Alternatively, we might make X the domain and $Y \times Z$ the codomain:

$$R : X \to Y \times Z = \{x \mapsto (y, z) \mid P(x, y, z)\} \tag{2.4.9}$$

In what follows, the term *relation* means binary relation, unless indicated otherwise.[9]

As in the case of relations *Unit Circle* and '≤' above, it is possible for the domain and codomain of a relation to be the same set, in which case the relation is **homogeneous on** its domain. Homogeneous relations have several properties not shared by inhomogeneous relations. Despite this, if it is useful to do so, we may always extend a relation to be homogeneous on the union of its domain and codomain.

2.4.3 SPECIAL RELATIONS

A *general* relation has no restrictions: elements of the domain and codomain may appear in ordered pairs once, several times, or not at all. If each element of the domain occurs at most once, then each element of the domain associates with a single value in the codomain. Such a relation is called a **function**, and its codomain is called its **range**. This is an extension of the familiar idea of a function, such as sine or cosine: given a value in the domain, it yields a *unique* value in the codomain. This notion of function also corresponds closely to the idea of function in a programming language. The *Has Parent* relation of Figure 2.6 is *not* a function, but its converse, *Has Child*, in Figure 2.7, happens to be a function.

We may use infix notation to show that elements are related, for example, we may write *Jane Has Parent Mary* in just the same way that we would write $1 \leq 2$. We may also **apply** a relation to an argument using the dot notation familiar to programmers, e.g., $Jane.Has\ Parent = Mary$. In the case of a function, such as *Has Mother*, we may still write $Jane.Has\ Mother = Mary$, but we may also use the more familiar functional notation $Has\ Mother(Jane) = Mary$.

A Cartesian product is always associated with two or more (trivial) **projection functions**. For example, given the domain $X \times Y$,

$$\pi_x : X \times Y \to X = \{(x, y) \mapsto x\}$$
$$\pi_y : X \times Y \to Y = \{(x, y) \mapsto y\}$$

We may extend this idea to relations. For example, given

$$Has\ Parent = \{Peter \mapsto Mary, Peter \mapsto Mark, Mary \mapsto Jane, Mary \mapsto Paul\}$$

we have

$$\pi_{Children}(Has\ Parent) = \{(Peter, Mary) \mapsto Peter, (Peter, Mark) \mapsto Peter, (Mary, Jane) \mapsto Mary, (Mary, Paul) \mapsto Mary\}$$
$$\pi_{Parents}(Has\ Parent) = \{(Peter, Mary) \mapsto Mary, (Peter, Mark) \mapsto Mark, (Mary, Jane) \mapsto Jane, (Mary, Paul) \mapsto Paul\}$$

The converse of a function isn't necessarily another function, but it is always a relation. For example, the relation that maps each integer onto its square is a

[9] The term 'relational database' refers to exactly the same notion of relation as described here.

function, $\{\ldots,-2 \mapsto 4,-1 \mapsto 1,0 \mapsto 0,1 \mapsto 1,2 \mapsto 4,\ldots\}$, but its inverse, $\{\ldots,0 \mapsto 0,1 \mapsto 1,1 \mapsto -1,4 \mapsto 2,4 \mapsto -2,\ldots\}$, is not. Similarly, although sin is a well-known trigonometric function, since $\sin^{-1}(0)$ has an infinite set of values, $\{\ldots,-2\pi,-\pi,0,\pi,2\pi,\ldots\}$, $\sin^{-1}$ isn't a function. In the case that the converse of a function *is* also a function, it is then called its **inverse**. If a function and its inverse are both total, they define a (one-to-one) **correspondence**. The function,

$$Successor : Integers \rightarrow Integers = \{x \mapsto y \mid y = x + 1\} \tag{2.4.10}$$

has the inverse,

$$Predecessor : Integers \rightarrow Integers = \{x \mapsto y \mid x = y + 1\} \tag{2.4.11}$$

and both are examples of correspondences; every integer has both a successor and a predecessor.

A **sequence** or **list**, such as $[a,b,a,c,b,d]$ (in which both the order and repetition of terms matter), is simply a function that maps positive integers onto its terms, i.e., the function,

$$\{1 \mapsto a,2 \mapsto b,3 \mapsto a,4 \mapsto c,5 \mapsto b,6 \mapsto d\} \tag{2.4.12}$$

A **bag** is like a sequence in that each element can appear more than once, but like a set in that the order of the elements is unimportant. For an example, think of your loose change. It is not only the set of coins that matters, but how many you have of each denomination. The bag {5¢, 5¢, 10¢, 20¢, 20¢, 20¢} can be expressed as the function $\{5¢ \mapsto 2, 10¢ \mapsto 1, 20¢ \mapsto 3\}$.

A relation R such that $\forall x \forall y (x \ R \ y \Rightarrow y \ R \ x)$ is said to be **symmetric**. The relations *Sibling Of* and *Married To* (with their everyday meanings) are symmetric. So is '=', because $\forall x \forall y (x = y \Rightarrow y = x)$.

If $(x \ R \ y \wedge y \ R \ x) \Rightarrow x = y$, then R is said to be **anti-symmetric**. The relation '$\leq$' is anti-symmetric, because $x \leq y \wedge y \leq x \Rightarrow x = y$. The relation '$<$' is vacuously anti-symmetric, because there is no case where $x < y \wedge y < x$.

A relation R that (among other things) maps each element to itself, so that $x \ R \ x$, for all x, is said to be **reflexive**. The relation '$\leq$' is reflexive, because $\forall x (x \leq x)$. The relation '$<$' is **irreflexive** because there is no case where $x < x$.

A relation R such that $\forall x \forall y \forall z (x \ R \ y \wedge y \ R \ z \Rightarrow x \ R \ z)$ is said to be **transitive**. The relation '$\leq$' is transitive, because $x \leq y \wedge y \leq z \Rightarrow x \leq z$. The relation '*Part Of*' is transitive: if a spoke is part of a wheel and the wheel is part of a bicycle, then the spoke is part of the bicycle.

A **partial ordering** is a relation that is reflexive, anti-symmetric, and transitive. The subset relation is a paradigm of a partial order: $S \subseteq S$, $(S \subseteq T \wedge T \subseteq S) \Rightarrow S = T$, and $(S \subseteq T \wedge T \subseteq U) \Rightarrow S \subseteq U$. For example, the subset relation on the subsets of {*Peter*, *Paul*, *Mark*} is clearly transitive:

$$\{Peter\} \subseteq \{Peter, Paul\} \subseteq \{Peter, Paul, Mark\}$$
$$\Rightarrow \{Peter\} \subseteq \{Peter, Paul, Mark\}$$

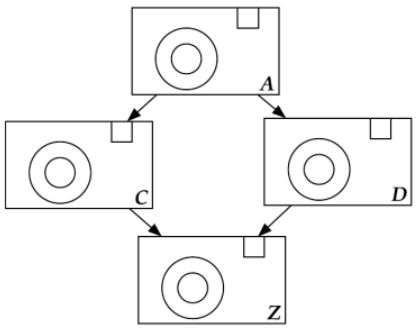

FIGURE 2.8

A comparison between cameras. Camera *C* has more pixels than *D*, but *D* has zoom and flash. *C* and *D* aren't comparable. Camera *A* has as many pixels as *C* and has zoom and flash like *D*. It is superior to both. *Z* has only as many pixels as *D* and like *C*, no zoom or flash. It is inferior to both.

Not every pair of elements in a partial order needs to be comparable. For example, neither $\{Peter, Mark\} \subseteq \{Paul, Mark\}$ nor $\{Paul, Mark\} \subseteq \{Peter, Mark\}$ is true. As a further example, consider a comparison between cameras. Camera *C* may have more pixels than camera *D*, but may not have as many features, so we cannot (in general) say that one is better than the other. In contrast, camera *A* may be superior to both *C* and *D* in both pixels and features, while camera *Z* may be inferior to both. (See Figure 2.8.)

2.4.4 OPERATIONS ON RELATIONS

It is possible to define many useful operations that can combine binary relations. These form an **algebra of relations**. The reader should note that the *algebra of relations* is distinct from the relational algebra associated with databases — which typically involves higher-order relations.

Since relations are sets, we can perform all the usual set operations. In addition, given a relation from X to Y, we can find the subset of y values associated with a given value of x, the converse relation from Y to X, etc.

If two relations share the same domain and codomain, we can find their union, intersection, or difference simply by applying these operations to their sets of pairs. Similarly, we can speak of one relation being a subset of another, or of two relations overlapping, or being disjoint.

If the codomain of relation Q is also the domain of a second relation R, we may form their **relational product** (denoted by **;**) defined as follows[10]:

$$Q\,;R = \{x \mapsto z \mid x\ Q\ y \wedge y\ R\ z\} \tag{2.4.13}$$

Consider the following two relations:

$$Sqrt = \{\ldots,0 \mapsto 0,1 \mapsto 1,1 \mapsto -1,4 \mapsto 2,4 \mapsto -2,\ldots\}$$
$$Seq = \{1 \mapsto a,2 \mapsto b,3 \mapsto a,4 \mapsto c,5 \mapsto b,6 \mapsto d\}$$

Since the codomain of *Sqrt* and the domain of *Seq* below are both integers, it is possible to form their product:

$$Sqrt\,;Seq = \{1 \mapsto a,4 \mapsto b,9 \mapsto a,16 \mapsto c,25 \mapsto b,36 \mapsto d\}$$

(Values in the codomain of *Sqrt*, such as -2, that have no corresponding element in the domain of *Seq*, do not contribute to the product.)

A **homogeneous relation** has the same domain and codomain ($dom(R) = codom(R)$), so it is possible for it to form a product with itself. Such products are written as powers:

$$R^2 = R\,;R$$
$$R^3 = R\,;R\,;R$$

By extension, we write

$$R^1 = R \tag{2.4.14}$$

and even

$$R^0 = \{x \mapsto x \mid x \in dom(R)\} \tag{2.4.15}$$

to denote the **identity relation**, $I_{dom(R)}$, **on** the domain of R.

A **closure of a relation** is formed by adding just enough pairs to the relation to give it some desired property:

- The **symmetric closure** of homogeneous relation R is the union of R with its converse, i.e., $R \cup R^{-1}$. (If $x \mapsto y$ is a pair in R, then $y \mapsto x$ is a pair in R^{-1}.)
- The **reflexive closure** of homogeneous relation R is the union of R with its corresponding identity, i.e., $R \cup R^0$. (Since R^0 is reflexive, so is the union.)
- The **transitive closure**, R^+, of homogeneous relation R is defined by the infinite union

  $$R^+ = R^1 \cup R^2 \cup R^3 \cup \cdots \tag{2.4.16}$$

  (If $x \mapsto y$ and $y \mapsto z$ are pairs in R, then $x \mapsto z$ is a pair in R^2, and so on for R^3, etc.)

[10] The reader may wish to pronounce '**;**' as 'followed by.'

- Finally, the **reflexive transitive closure**, R^*, of homogeneous relation R is the union $R^+ \cup R^0$, alternatively defined by the infinite union

$$R^* = R^0 \cup R^1 \cup R^2 \cup R^3 \cup \cdots \tag{2.4.17}$$

Even though transitive closures are defined by infinite unions, if a relation has a finite domain, X, its closure must also be finite, being at most $X \times X$.[11]

To illustrate various operations on relations, consider the following relation[12]:

$$Daughter\ Of = \{c \mapsto p \mid c\ Has\ Parent\ p \wedge Female(c)\} \tag{2.4.18}$$

This is rather an awkward definition because it mixes relational notation and predicate calculus. In using relations, we prefer to use an identity relation, as follows:

$$Is\ Female = I_{Females} \tag{2.4.19}$$

where $I_{Females}$ is the identity relation on the set *Females*.

Relations 2.4.20 and 2.4.21 are subsets of the *Has Parent* relation, obtained by taking a subset of its domain or codomain, and are called **left-restrictions** and **right-restrictions**, respectively.

$$Daughter\ Of = Is\ Female\ ;\ Has\ Parent \tag{2.4.20}$$

$$Has\ Mother = Has\ Parent\ ;\ Is\ Female \tag{2.4.21}$$

With a similar definition, $Is\ Male = I_{Males}$, we can then proceed as follows:

$$\begin{aligned}
Has\ Child &= Has\ Parent^{-1} \\
Has\ Father &= Has\ Parent\ ;\ Is\ Male \\
Son\ Of &= Is\ Male\ ;\ Has\ Parent \\
Has\ Daughter &= Daughter\ Of^{-1} \\
Mother\ Of &= Has\ Mother^{-1} \\
Has\ Son &= Son\ Of^{-1} \\
Father\ Of &= Has\ Father^{-1} \\
Is\ Person &= Is\ Male \cup Is\ Female \\
Has\ Grandparent &= Has\ Parent^2 \\
Has\ Grandchild &= Has\ Child^2 = Has\ Grandparent^{-1} \\
Has\ Ancestor &= Has\ Parent^+ \\
Has\ Descendant &= Has\ Child^+ = Has\ Ancestor^{-1}
\end{aligned}$$

[11] Not all closures are finite. The $\leq$ relation is the infinite reflexive transitive closure of the *Successor* relation.

[12] The names of the relations in this section have self-explanatory meanings equivalent to the normal family relationships recognised by English speakers.

$$\mathit{Has\ Aunt} = (\mathit{Has\ Grandparent}\,;\mathit{Has\ Daughter}) \setminus \mathit{Has\ Mother} \tag{2.4.22}$$

(In Relation 2.4.22, your aunt is any daughter of your grandparents except your own mother.)

Since relations are special kinds of sets, all the axioms and theorems of set theory apply to relations. In addition, they have properties concerning products and converses:

$$(R\,;S)\,;T = R\,;(S\,;T) = R\,;S\,;T$$
$$R\,;(S \cup T) = (R\,;S) \cup (R\,;T)$$
$$(S \cup T)\,;R = (S\,;R) \cup (T\,;R) \tag{2.4.23}$$
$$(R^{-1})^{-1} = R \tag{2.4.24}$$
$$(R\,;S)^{-1} = S^{-1}\,;R^{-1}$$

However, the following are inequalities, except in special cases:

$$R\,;S \neq S\,;R \tag{2.4.25}$$
$$R^0 \subseteq R^1\,;R^{-1} \tag{2.4.26}$$
$$(S \cap T)\,;R \subseteq (S\,;R) \cap (T\,;R) \tag{2.4.27}$$
$$R\,;(S \cap T) \subseteq (R\,;S) \cap (R\,;T) \tag{2.4.28}$$

Inequality 2.4.25 reflects the fact that your mother's father isn't your father's mother. To illustrate inequality 2.4.26, your parent's child may be yourself, but it may also be your brother or sister.[13]

In the case of 2.4.27, consider

$$(\mathit{Has\ Father}\,;\mathit{Has\ Child}) \cap (\mathit{Has\ Mother}\,;\mathit{Has\ Child}),$$

which, when applied to you, yields yourself and your full brothers and sisters. In contrast, the expression (*Has Father* $\cap$ *Has Mother*) is an empty relation, and therefore so is the equivalent left-hand side of 2.4.27.

In the case of 2.4.28, consider

$$(\mathit{Has\ Ancestor}\,;\mathit{Has\ Child}) \cap (\mathit{Has\ Ancestor}\,;\mathit{Has\ Grandchild}),$$

which is certainly not empty, and when applied to you, yields yourself and your ancestors. But in a well-ordered society, the relation (*Has Child* $\cap$ *Has Grandchild*) — and thus the equivalent left-hand side of 2.4.28 — should be empty.

Finally, another potentially useful way to combine relations is by **parallel composition**:

$$Q \| R = \{x \mapsto (y, z) \mid x\ Q\ y \wedge x\ R\ z\} \tag{2.4.29}$$

Contrast *Has Parent* with *Has Father*$\|$*Has Mother*: The first is a relation from a child to both its father and mother; the second is a relation (which happens to be a function) that maps a child to a single (*Father*, *Mother*) pair.

---

[13] For this reason, and despite Equation 2.4.24, not everyone is happy with the notation R^{-1} to denote the inverse of R. Many prefer $R^{\leftarrow}$. Even so, such usage is well-established for $\sin^{-1} x$, $\cos^{-1} x$, etc.

2.5 GRAPHS AND SCHEMAS

Systems analysts love to draw diagrams. The virtue of diagrams is that they appeal to the eye. Because of the way the brain processes visual information, it is usually easier to assimilate the information they display than the same information given in algebraic or tabular form. Even so, diagrams are only useful when they are simple. An attempt to make a diagram useful for more than one purpose can sometimes make it useful for no purpose at all.

Many diagrams are actually mathematical objects called **graphs** or schemas.[14] A graph is a diagram that defines a relation. A *schema* shows a **meta-relation**: *a relationship between relations*.

2.5.1 GRAPHS

Most of us should be familiar with the idea of a graph that shows a relation between real numbers. Figure 2.5 shows the graph of the relation $x^2 + y^2 = 1$, describing the unit circle. We may use a similar approach to show a relation between integers. On the other hand, when we want to show the relation between *Parents* and *Children*, axes are less appropriate, and a different kind of graph is used. Such a graph consists of labelled **vertices** (usually drawn as small circles or rectangles) connected by **edges** (usually drawn as arrows). (See Figure 2.9.)[15] The edges are **directed** from the domain to the codomain. In other words, an edge **from** u **to** v means that the pair $u \mapsto v$ is a member of the relation. As a result, there may be at most one edge from u to v. The edge has **initial vertex** u and **terminal vertex** v. We denote an edge from u to v by $u \rightarrow v$. The edge $u \rightarrow v$ is said to be an **out-edge** of u and an **in-edge** of v. The number of edges entering v is called its **in-degree**, and the number of edges leaving v is called its **out-degree**. If we want to emphasise that the edge $u \rightarrow v$ belongs to a particular graph (or relation) G, we write $u \underset{G}{\rightarrow} v$. Because their edges are directed, such graphs are called directed graphs, or **digraphs**.

A **subgraph** H of graph G comprises some subset of the vertices of G and some subset of the edges of G, subject to the sensible restriction that the edges of H must link vertices that belong to H. (See Figure 2.10.) H is therefore a graph of a subset of the pairs of the relation represented by G.

Sometimes a diagram forms a **bi-partite graph**. This is a graph with two kinds of vertices and two kinds of edges. The edges link vertices of different kinds, but never of the same kind. For example, Figure 2.11 is a bi-partite graph showing the relationship between some processes (rectangles) and files (circles). It contains *two* subgraphs that define *two* relations, an *Output* relation from processes to files, and an

[14] Strictly speaking, since I consistently use the Latin-derived plurals, 'matrices', 'vertices', and 'indices', rather than 'matrixes', 'vertexes', and 'indexes', I should say 'schemata' rather than 'schemas'. But I don't. 'Schema' is derived from *Greek*, and there I draw the line.

[15] We are distinguishing here between Mark and Mary as *Parents* and Mark and Mary as *Children*. This isn't perhaps the best way to express their family relationships. Figure 2.12 expresses it better.

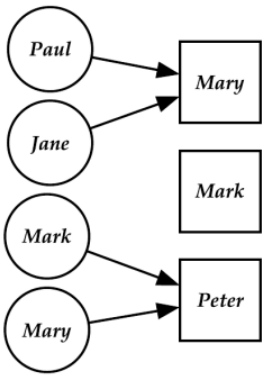

FIGURE 2.9

Graph of the *Has Child* relation from the domain *Parents* (circles) to the codomain *Children* (squares). The edges represent ordered pairs.

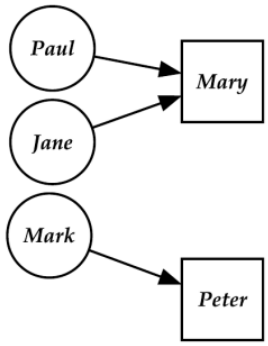

FIGURE 2.10

A subgraph of the *Has Child* relation of Figure 2.9.

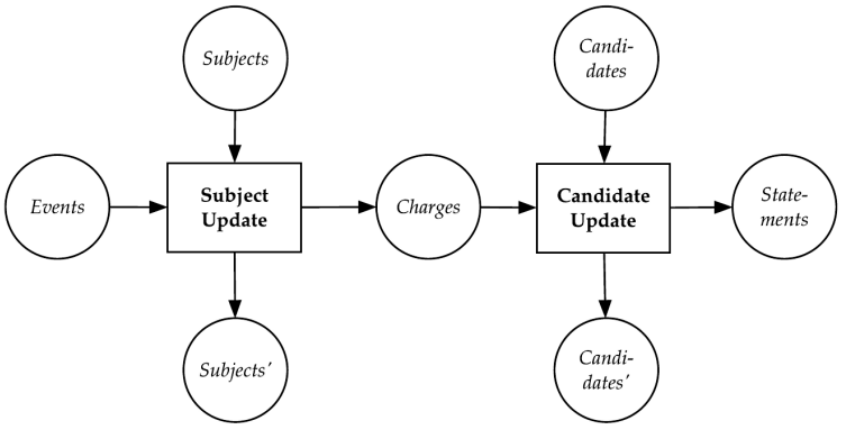

FIGURE 2.11

A bi-partite graph, showing both an *Output* relation (rectangles to circles) and an *Input* relation (circles to rectangles).

Input relation from files to processes. The *Output* relation is given by the edges from rectangles to circles, and the *Input* relation by the edges from circles to rectangles.

Similar conventions are used to draw homogeneous relations, where the domain and codomain are identical, although there now is only one kind of vertex. If we

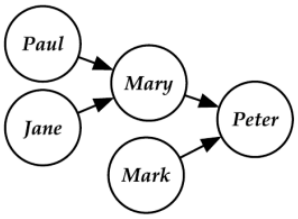

FIGURE 2.12

The *Has Child* relation on *Persons*. Compare this with Figure 2.9, where the *Mary* vertex appears twice.

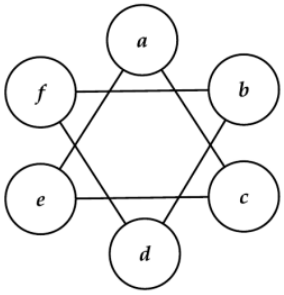

FIGURE 2.13

An undirected graph. Each edge is equivalent to a pair of directed edges with opposite directions.

consider the *Has Child* relation on *Persons* (the union of *Parents* and *Children*), we obtain the homogeneous graph of Figure 2.12.

If both $u \mapsto v$ and $v \mapsto u$ are both pairs of a relation, we can draw two edges, $u \rightarrow v$ and $v \rightarrow u$, or we can draw an arrowhead at both ends of the same edge. Conventionally, if a relation is *known* to be symmetric ($u \rightarrow v \Rightarrow v \rightarrow u$), rather than draw two arrowheads on every edge, we can omit both of them (see Figure 2.13). We also write $v \leftrightarrow u$, rather than both $u \rightarrow v$ and $v \rightarrow u$. Such graphs are called **undirected**.

An edge from a vertex to itself is called a **loop** (e.g., the edge $a \rightarrow a$ in Figure 2.14). A reflexive relation would have a loop on every vertex, so if a relation is *known* to be reflexive, we usually omit the loops to avoid clutter.

A **path** is a sequence of edges such that the terminal vertex of each edge (except the last) is the initial vertex of the next. The **length** of a path is the number of edges in the path. In Figure 2.14 on the facing page, $a \rightarrow c \rightarrow e \rightarrow f$ is a path of length 3 from a to f, and $a \rightarrow b \rightarrow f$ is a **parallel path** of length 2.

If any path leads from u to v, u is **connected** to v. The notation $u \rightarrow^2 v$ means that u is connected to v by a path of length 2, $u \rightarrow^+ v$ means that u is connected to v by a path of at least one edge, and $u \rightarrow^* v$ means that u is connected to v by a path of any length, including zero. Clearly, $v \rightarrow^* v$ is true for every vertex v of a homogeneous graph.

A path that has the same initial and terminal vertex and at least two edges is called a **cycle**.[16] If $u \rightarrow^+ v \rightarrow^+ u$, where $v \neq u$, then a cycle passes through vertices v and u. (In Figure 2.14 the path $a \rightarrow c \rightarrow e \rightarrow a$ is a cycle passing through a, c, and e.) A graph that contains at least one cycle is said to be **cyclic**, otherwise it is said to be **acyclic**.

[16] Some authors count loops as cycles, too.

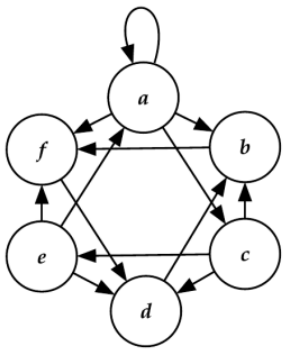

FIGURE 2.14

A homogeneous directed graph. The edge $a \to a$ is a loop. The path $a \to c \to e \to a$ is a cycle.

A **directed acyclic graph** is often referred to as a **DAG**. If $u \to^* v$ then v is a **successor** of u, and u is an **antecedent** of v. Any finite DAG must contain at least one vertex that has no antecedent. Such a vertex is called an **initial vertex** of the graph.[17] Likewise, a finite DAG must contain at least one vertex with no successor. Such a vertex is called a **final vertex**.

Every DAG represents a partial ordering. Every DAG and every partial order has at least one topological sort. A **topological sort** of a graph is a sequence of its vertices, such that, for all edges $u \to v$, u precedes v in the sequence. Consider the $\subset$ relation on $2^{\{a,b,c\}}$, i.e., the domain

$$\{\{a,b,c\},\{a,b\},\{b,c\},\{a,c\},\{a\},\{b\},\{c\},\{\}\}.$$

(See Figure 2.15 on the next page.) Then the sequences

$$[\{\},\{a\},\{b\},\{a,b\},\{c\},\{b,c\},\{a,c\},\{a,b,c\}]$$
$$[\{\},\{c\},\{b\},\{a\},\{a,c\},\{b,c\},\{a,b\},\{a,b,c\}]$$

are two of its 48 possible topological sorts.

A simple method for finding a topological sort is given by Algorithm 2.1.

The **transitive closure** G^+ of graph G is the graph of the transitive closure R^+ of its underlying relation R. Therefore, for example, if $u \to v \to w \to x$ is a *path* in G, then $u \to x$ is an *edge* in G^+. (See Figure 2.16.) (The symmetric, reflexive, and reflexive transitive closures of G are defined in similar ways.)

[17] The *Successor* relation on the integers forms an infinite DAG, but it has no initial vertex because there is no smallest number.

Algorithm 2.1 A simple method for finding a topological sort of a DAG.

1. Begin with an empty sequence, S, and a copy G of the graph.
2. Choose any initial vertex, V of G, i.e., any vertex with no in-edges.
3. Append V to S.
4. Delete V and all its out-edges from G.
5. Repeat Steps 2–4, until no more initial vertices remain in G.
6. (If any vertices remain in G, the original graph was not acyclic.)

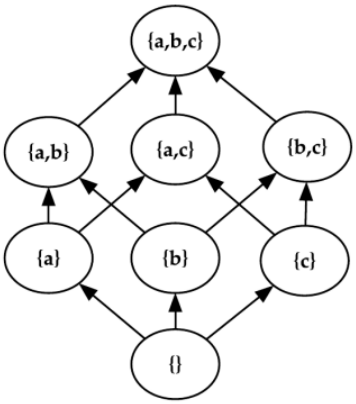

FIGURE 2.15

A DAG representing the $\subset$ relation on the powerset of $\{a,b,c\}$.

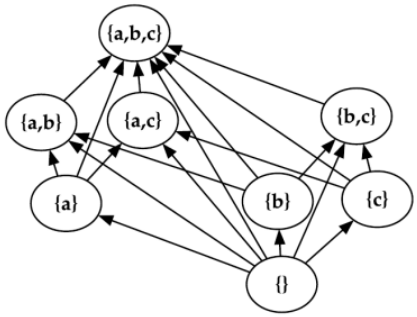

FIGURE 2.16

The transitive closure of the $\subset$ relation on the powerset of $\{a,b,c\}$.

A **transitive reduction** of graph G is any subgraph $H \subseteq G$ such that H and G have the same transitive closure ($H^+ = G^+$). If no edge can be removed from H without destroying the relation $H^+ = G^+$, then H is a **minimal transitive reduction**. If G is acyclic, its minimal transitive reduction is unique and is called its **transitive root**.

Transitive reductions are useful when we want to present the graph of a transitive relation with minimum clutter. Contrast Figure 2.16 (with 19 edges), which shows the transitive closure of the $\subset$ relation on the subsets of $\{a,b,c\}$, with its transitive root (with 12 edges), which is shown in Figure 2.15.

2.5.2 CONNECTED COMPONENTS

In a cyclic graph, if both $u \rightarrow^* v$ and $v \rightarrow^* u$, then u and v are said to be **strongly connected**. Clearly, every vertex is strongly connected to itself, and every vertex that is part of a cycle is strongly connected to every other vertex in the cycle.

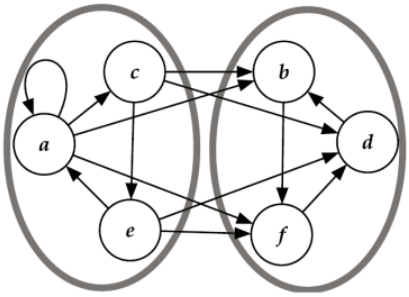

FIGURE 2.17

The graph of Figure 2.14 rearranged to reveal its strongly connected components. Edges lead from the left to the right component, but no edges lead from the right to the left component.

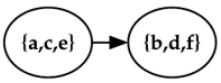

FIGURE 2.18

The reduced graph of the strongly connected components of the graph of Figure 2.14.

A **strongly connected component** of a graph is a set of vertices C, such that,

$$\forall u \forall v (u \in C \wedge v \in C \Rightarrow u \rightarrow^* v \wedge v \rightarrow^* u) \tag{2.5.1}$$

In other words, at least one cycle passes through every vertex within a strongly connected component. From now on, when we talk about a strongly connected component, we mean a *maximal* strongly connected component, one to which no more strongly connected vertices could be added.

The strongly connected components of a graph partition the vertices of the graph; every vertex belongs to exactly one strongly connected component. Figure 2.17 shows the same graph as Figure 2.14, but the vertices have been rearranged. The grey ellipses enclose its strongly connected components, $\{a, c, e\}$ and $\{b, d, f\}$.

Every directed graph G has a corresponding **reduced graph** H. Each vertex of H is a *set* of vertices of G that belong to the same (maximal) strongly connected component. The edges of H are the sets of edges of G that link different components. Clearly, a reduced graph must be acyclic; if a cycle existed between two strongly connected components, they could not be maximal. Figure 2.18 shows the reduced graph of Figure 2.14.

A cyclic graph has several minimal transitive reductions, because the vertices within its strongly connected components may be connected in any order while still preserving the same closure. In contrast, its reduced graph, being acyclic, has a unique transitive root.

In an *undirected* graph, if u is connected to v, then v is connected to u, so u and v must be strongly connected. The strongly connected components of an undirected graph are known simply as **connected components**. There can be no edges between its connected components, which therefore form disjoint subgraphs. As an example,

think of the towns in a group of islands, connected by roads. Assuming no roads cross between the islands, the islands enclose connected components. Building a road bridge between towns on two different islands would merge two components into one; it would *not* (by definition) create an edge between components. In the case of the undirected graph of Figure 2.13, the connected components are $\{a,c,e\}$ and $\{b,d,f\}$. A single new edge between a and b would result in one component, $\{a,b,c,d,e,f\}$. If a graph has only one connected component, the graph itself is said to be **connected**.

Although heterogeneous graphs do not share the interesting and useful closure properties of homogeneous graphs, a bi-partite graph such as that of Figure 2.11 (page 45) — which displays an *Input* relation from files to processes and an *Output* relation from processes to files — can be expressed as two homogeneous products: *Feeds* = *Output* ; *Input* on processes, and *Determines* = *Input* ; *Output* on files. Thus, we can say that the *Subject Update* process *Feeds* the *Candidate Update* process, and that the data in the *Events* and *Subjects* files *Determines* the values in the *Charges* and *Subjects'* files and transitively *Determines* the *Statements* and *Candidates'* files. We shall make good use of these relations in Chapter 7.

2.5.3 ROOTED TREES

A **tree** is a connected undirected acyclic graph. It follows that a tree has the minimum number of edges to connect its vertices; if the tree has V vertices, it has exactly $(V-1)$ edges.

In contrast, a **rooted tree** is directed. It has a special vertex called the **root**. The vertices of a tree are called **nodes**; its edges are called **branches**. Implicitly, the branches are normally considered to be directed *away* from the root.

Usually, a tree is drawn with its root at the top, with the nodes ordered downwards according to their distances from the root. (See Figure 2.19 on the facing page.) Given any branch, the node closer to the root is called the **parent node**, and the node further from the root is called the **child node**. It is a property of rooted trees that every child node has exactly one parent, but a parent node may have any number of children. The number of its children is called its **degree**. The mapping from child to parent defines the *Parent* function, but *Child* is a one-to-many relation, where $Child = Parent^{-1}$. A **descendant** is any node in the transitive closure of the *Child* relation; an **ancestor** is any node in the closure of the *Parent* function. A node with no children is called a **leaf**, or **external node**. All other nodes are called **internal nodes**.

Any node V of a tree, its descendants, and their associated edges form a **sub-tree** of the tree. V is called the root of the sub-tree.

The **height** of a tree or sub-tree is 1 plus the length of the longest path from its root to any descendant.[18] The total number of nodes in the tree or sub-tree is called its **weight**.

[18] We say that a degenerate tree consisting only of a root node has height 1. Some authors say that such a tree has height zero, a privilege we reserve for the empty tree. The distinction is immaterial here.

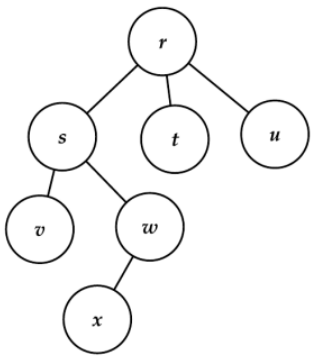

FIGURE 2.19

A rooted tree. Node r is the root; t, u, v, and x are leaves; s and w are internal nodes. The sub-tree rooted at s contains s, v, w, and x.

2.5.4 LABELLED GRAPHS

The vertices or the edges of a graph may be **labelled** with numbers or other symbols.

Although the vertices of a graph are normally labelled with domain values, it is possible to add further labels to them, representing the values of one or more functions of the vertices. For example, a graph of airline connections might show the time zone of each airport.

When the edges are labelled, the graph then represents a ternary relation. In a labelled graph, it isn't unusual to have more than one edge from vertex u to vertex v, provided they have different labels. For example, a labelled graph may contain both the edges $u \underset{A}{\rightarrow} v$ and $u \underset{B}{\rightarrow} v$. Labelled graphs may be used for three different purposes.

First, a labelled graph might represent a function from *pairs* of vertices to labels, for example, the vertices might represent airports, and the labels might represent the great circle distances between them. (See Figure 2.20 on the next page.) In such a case, there can be at most one edge in each direction between any two vertices.

Second, a labelled graph might represent a function where each edge maps its label and source vertex to its target vertex. In such a case, at most one edge may leave each vertex with a given label, although several (differently labelled) edges could connect the same pair of vertices. (See Figure 2.21.)

Third, and equivalently, a labelled graph may represent a *set of relations*, whose names are given by the labels on the edges. For example, the edge $u \underset{A}{\rightarrow} u$ means that (u,v) is an element of relation A, and the edge $v \underset{B}{\rightarrow} w$ means that (v,w) is an element of relation B. The graphs of these component relations are therefore simply the subgraphs we obtain by considering only the correspondingly labelled edges of

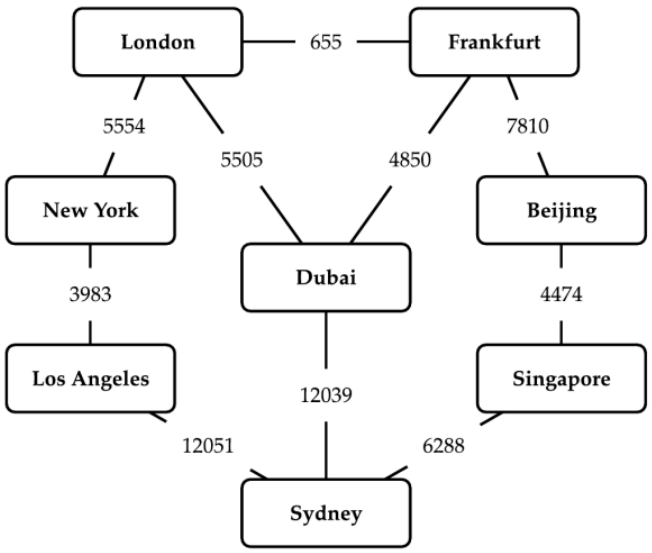

FIGURE 2.20

A labelled graph, representing a function from pairs of vertices to labels. It gives the distances in kilometres between pairs of international airports.

the graph. Conversely, we may superimpose the graphs of several relations, provided we label the edges properly. (See Figure 2.22 on the facing page.)

(A bi-partite graph, such as Figure 2.11, is a special instance of this practice, where the two sets of edges (*Input* and *Output*) can be distinguished by their direction, so they need no labels.)

2.5.5 MATRIX REPRESENTATION OF GRAPHS

It can be useful to represent a graph (and therefore its underlying relation) as an **adjacency matrix**. Each row of the matrix represents a source vertex, and each column represents a destination vertex. Each cell of the matrix therefore represents a potential edge, from the row to the column. If the graph is labelled, we can write the label of each edge into its cell; if it is unlabelled, we can mark the edges with any symbol we choose.

Although graphs are useful for tutorial purposes, the graphs of real-world problems are often too complex to be easily understood, and finding their products and closures (for example) by eye can be almost impossible. Matrices, on the other hand, are easy to manipulate within a computer.

The adjacency matrix of the graph of Figure 2.14 is given in Table 2.1. The entries in row a represent the edges $a \to a$, $a \to b$, $a \to c$, and $a \to f$.

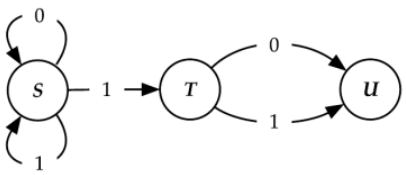

FIGURE 2.21

A graph whose edges are labelled 0 and 1. There are pairs of differently labelled edges from T to U and from S to S.

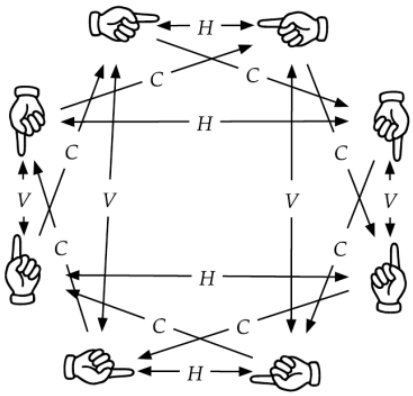

FIGURE 2.22

A labelled graph representing some transformations of a simple figure. The C relation is clockwise rotation, V is vertical reflection, and H is horizontal reflection. V and H are reflexive. C^{-1} is anticlockwise rotation. Cycles and parallel paths reveal interesting properties, such as $H\,;C = C\,;V$.

Table 2.1 The adjacency matrix of the graph of Figure 2.14

	a	b	c	d	e	f
a	X	X	X	–	–	X
b	–	–	–	–	–	X
c	–	X	–	X	X	–
d	–	X	–	–	–	–
e	X	–	–	X	–	X
f	–	–	–	X	–	–

Table 2.2 The adjacency matrix of the reverse of Figure 2.14

	a	b	c	d	e	f
a	X	–	–	–	X	–
b	X	–	X	X	–	–
c	X	–	–	–	–	–
d	–	–	X	–	X	X
e	–	–	X	–	–	–
f	X	X	–	–	X	–

To reverse all the edges of the graph of Figure 2.14 and thus find its converse, we exchange rows and columns in Table 2.1, giving Table 2.2.

To find the adjacency matrix of the union of two relations, we mark each cell that is marked in *either* of their matrices. To find their intersection, we mark each cell that is marked in *both* their matrices. In similar style, we can find the asymmetric difference $R \setminus S$ by deleting all entries from R that are marked in S.

Table 2.3 The adjacency matrix of the symmetric closure of Figure 2.14

	a	*b*	*c*	*d*	*e*	*f*
a	X	X	X	–	X	X
b	X	–	X	X	–	X
c	X	X	–	X	X	–
d	–	X	X	–	X	X
e	X	–	X	X	–	X
f	X	X	–	X	X	–

Table 2.4 The adjacency matrix of the reflexive closure of Figure 2.14

	a	*b*	*c*	*d*	*e*	*f*
a	X	X	X	–	–	X
b	–	X	–	–	–	X
c	–	X	X	X	X	–
d	–	X	–	X	–	–
e	X	–	–	X	X	X
f	–	–	–	X	–	X

The union of Table 2.1 and Table 2.2 is shown in Table 2.3. It represents the symmetric closure of Figure 2.14, i.e., the undirected graph with the same edges. (Not surprisingly, the matrix is symmetrical about its leading diagonal.)

To create the matrix of the reflexive closure, we simply mark each cell along the leading diagonal of the matrix, creating a loop on each vertex, as in Table 2.4.

We can find the relational product $(R\,;S)$ of relations $R : U \to V$ and $S : V \to W$ using adjacency matrices. Suppose $u \to v$ is an edge in R and $v \to w$ is an edge in S. Then $u \to w$ is an edge in $R\,;S : U \to W$. Suppose we want to discover if edge $u \to w$ is in $R\,;S$. If it is, then $u \to v \to w$, for some v. We therefore scan row u of R and column w of S for each potential value of V. If both (u,v) and (v,w) are marked, then we mark (v,w) in the matrix of the product. Clearly, this procedure has complexity $O(\mid U \mid \times \mid V \mid \times \mid W \mid)$.

R and S don't need to be distinct. Suppose we want to find all paths of length 2 in Figure 2.14 on page 47. Beginning with the adjacency matrix of Table 2.1, we construct Table 2.5, as just described. To help the reader understand the method, the cells of Table 2.5 are marked with the vertex or vertices that complete the path. For example, there are two paths from c to f: $c \to b \to f$ and $c \to e \to f$.

Finally, consider the problem of finding the strongly connected components of Figure 2.14. Although rearranging the graph as in Figure 2.17 makes the solution obvious, not everyone finds it easy to visualise such a rearrangement. Since every vertex in a strongly-connected component has a path to every other vertex in the component, the reflexive transitive closure of the graph highlights the components. A drawing of the closure would contain many edges and would confuse the eye. Instead, we manipulate the matrix.

One way to find the closure would be to find the union of all paths of lengths $1,2,\ldots,(V-1)$ where V is the number of vertices. (No path can have more than $(V-1)$ edges without doubling back on itself.) This would have complexity $O(V^4)$. Fortunately, a more efficient method, called **Warshall's Algorithm**, has complexity $O(V^3)$. It works by modifying the matrix *in situ*.

Table 2.5 The matrix of all paths of length 2 in Figure 2.14

	a	*b*	*c*	*d*	*e*	*f*
a	*a*	*a*,*c*	*a*	*c*, *f*	*c*	*b*
b	–	–	–	*f*	–	–
c	*e*	*d*	–	*e*	–	*b*,*e*
d	–	–	–	–	–	*b*
e	*a*	*a*,*d*	*a*	*f*	–	*a*
f	–	*d*	–	–	–	–

Table 2.6 The effect of considering paths through vertex *a*

	a	*b*	*c*	*d*	*e*	*f*
a	X	X	X	–	–	X
b	–	X	–	–	–	X
c	–	X	X	X	X	–
d	–	X	–	X	–	–
e	X	X	X	X	X	X
f	–	–	–	X	–	X

Table 2.7 The effect of considering paths through vertices *a* and *b*

	a	*b*	*c*	*d*	*e*	*f*
a	X	X	X	–	–	X
b	–	X	–	–	–	X
c	–	X	X	X	X	X
d	–	X	–	X	–	X
e	X	X	X	X	X	X
f	–	–	–	X	–	X

We begin by marking the diagonal of the matrix, forming its reflexive closure. The result was already shown in Table 2.4. (If we were forming the *transitive closure* rather than the reflexive transitive closure, we would omit this step.)

We now reason as follows: any vertex that has a path *to* vertex *a* must also have a path to every vertex that has a path *from a*. Therefore, we copy the non-blank entries in *row a* into every row that has an entry in *column a*. In fact, *e* is the only vertex other than *a* that has an edge leading to *a*. The result is shown in Table 2.6.

We continue to reason in the same way: any vertex that has a path to vertex *b* must have a path to every vertex that can be reached from *b*. Therefore, we copy the entries in *row b* into every row that now has an entry in *column b*. In this case, *f* is the only vertex that doesn't. The result is shown in Table 2.7.

We continue the iteration through vertices *c*, *d*, *e*, and *f* in turn. The result is shown in Table 2.8.

The procedure we have just described is guaranteed to find such a **connectivity matrix** in just one iteration through intermediate vertices. The reader should pay attention to the order of the iterations. It is essential that the intermediate vertices (the ones the paths pass through) are considered in the outermost iteration, otherwise

Table 2.8 The effect of considering paths through all vertices *a* to *f*

	a	*b*	*c*	*d*	*e*	*f*
a	X	X	X	X	X	X
b	–	X	–	X	–	X
c	X	X	X	X	X	X
d	–	X	–	X	–	X
e	X	X	X	X	X	X
f	–	X	–	X	–	X

Table 2.9 The effect of grouping vertices with equal closures

	a	*c*	*e*	*b*	*d*	*f*
a	X	X	X	X	X	X
c	X	X	X	X	X	X
e	X	X	X	X	X	X
b	–	–	–	X	X	X
d	–	–	–	X	X	X
f	–	–	–	X	X	X

the algorithm will need to be applied repeatedly until the result is stable. Warshall's Algorithm can also be expressed independently of the matrix representation, as in Algorithm 2.2 below.

Algorithm 2.2 Warshall's Algorithm for finding the transitive closure of graph G.

1. Initialise a set of edges E to equal the set of edges in G.
2. If the *reflexive* transitive closure is required, add the edge $V \rightarrow V$ to E, for each vertex V of the graph.
3. For each vertex V of G,
   - For each pair of vertices U and W (not necessarily distinct),
     - If $U \rightarrow V$ and $V \rightarrow W$ are both members of E, add the edge $U \rightarrow W$ to E.
4. E is the set of edges in the transitive closure of G.

Notice that Table 2.8 contains only two different kinds of row: row a, c, or e, and row b, d, or f. Once we group like rows together, we have found the strongly connected components. Every vertex in a strongly connected component has a path to every other vertex in it and therefore to every vertex that can be reached from it. As a result, they have identical matrix entries. Reordering the rows and columns representing the vertices, as in Table 2.9, makes this clear.

Finally, by grouping vertices a, c, and e as the set $\{a,c,e\}$ and vertices b, d, and f as the set $\{b,d,f\}$, we obtain, in Table 2.10, the adjacency matrix of the closure of the reduced graph, shown in Figure 2.18. We can therefore find the reduced graph H of the strongly connected components of a directed graph G using Algorithm 2.3.

Finally, we can use matrix methods to find the transitive root of an acyclic graph G: we first find its transitive closure G^+ and then the product $G\,;G^+$, using methods

Algorithm 2.3 Finding the reduced graph H of the strongly connected components of G.

1. Find the transitive closure G^* of G (e.g., using Warshall's Algorithm).
2. Form the maximal sets of vertices in G^* that share the same closure. These sets are the strongly connected components of G. Create a vertex of the reduced graph H to correspond to each component of G.
3. If there is any edge from vertex V in component S to vertex U in component T in graph G, where S and T are different strongly connected components of G, then create an edge from the corresponding vertex S to vertex T of H.

Table 2.10 The connectivity matrix of the closure of the reduced graph of Figure 2.18

	$\{a,c,e\}$	$\{b,d,f\}$
$\{a,c,e\}$	X	X
$\{b,d,f\}$	–	X

just explained. The expression $G\,;G^+$ gives the set of all paths of length 2 or more in G and therefore includes any compound paths already present in G. Finding the asymmetric difference $G \setminus (G\,;G^+)$ eliminates all but the simple paths in G.

2.5.6 SCHEMAS

A **schema**, as we use the term here, is a collection of domains and the relations between them. There are three kinds of schema: an abstract schema, which is a mathematical object, a conceptual schema, which is used to design the structure of a database, and a physical schema, which is the embodiment of a schema in software.

A schema may be drawn as a labelled graph that shows, not a relation between elements of sets, but a meta-relation between whole sets. The vertices of a schema represent the domains or codomains of relations. The edges of a schema are labelled with the names of relations. A labelled directed edge $S \xrightarrow[R]{} T$ means that R is a relation with domain S and codomain T.

It is often important to distinguish functions and correspondences from more general relations. Different authors have used just about every conceivable convention

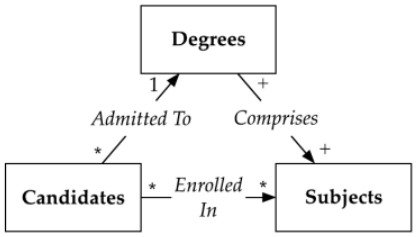

FIGURE 2.23

An example of a schema. *Candidates* are *Admitted To* exactly one *Degree* and may be *Enrolled In* any number of *Subjects*. Each *Degree Comprises* at least one *Subject*, and every *Subject* must be part of at least one *Degree*. *Admitted To* is a **total into** function, *Enrolled In* is a **partial into** relation, and *Comprises* is a **total onto** relation.

to show these distinctions.[19] Here, we label the ends of the edges according to the following conventions:

?: at most one element of the domain or codomain,

1: exactly one element of the domain or codomain,

$*$: zero or more elements of the domain or codomain,

+: one or more elements of the domain or codomain.

For example, Figure 2.23 represents the schema

$$Candidates \xrightarrow[Admitted\ To]{*\quad 1} Degrees$$

$$Candidates \xrightarrow[Enrolled\ In]{*\quad *} Subjects$$

$$Degrees \xrightarrow[Comprises]{+\quad +} Subjects$$

If the *destination* of an edge (the arrowhead) is labelled '1' or '+', this implies that the relation is total, because every element of its *source* maps to at least one element in the codomain. On the other hand, if the destination is labelled '?' or '$*$',

[19] Except, perhaps, the one used here. For example, the Z notation for program specification uses a variety of arrows that seem — to me at least — rather arbitrary and that still fail to cover all 16 possibilities.

Table 2.11 All 16 possible ways of labelling the edges of a schema

$S \xrightarrow[F]{? \quad ?} T$	$F : S \to T$ is a partial correspondence from S into T.
$S \xrightarrow[F]{? \quad 1} T$	$F : S \to T$ is a total correspondence from S into T.
$S \xrightarrow[R]{? \quad *} T$	$R : S \to T$ is the converse of a partial function from T into S.
$S \xrightarrow[R]{? \quad +} T$	$R : S \to T$ is the converse of a partial function from T onto S.
$S \xrightarrow[F]{1 \quad ?} T$	$F : S \to T$ is a partial correspondence from S onto T.
$S \xrightarrow[F]{1 \quad 1} T$	$F : S \to T$ is a total correspondence from S onto T.
$S \xrightarrow[R]{1 \quad *} T$	$R : S \to T$ is the converse of a total function from T into S.
$S \xrightarrow[R]{1 \quad +} T$	$R : S \to T$ is the converse of a total function from T onto S.
$S \xrightarrow[F]{* \quad ?} T$	$F : S \to T$ is a partial function from S into T.
$S \xrightarrow[F]{* \quad 1} T$	$F : S \to T$ is a total function from S into T.
$S \xrightarrow[R]{* \quad *} T$	$R : S \to T$ is a partial relation from S into T.
$S \xrightarrow[R]{* \quad +} T$	$R : S \to T$ is a total relation from S into T.
$S \xrightarrow[F]{+ \quad ?} T$	$F : S \to T$ is a partial function from S onto T.
$S \xrightarrow[F]{+ \quad 1} T$	$F : S \to T$ is a total function from S onto T.
$S \xrightarrow[R]{+ \quad *} T$	$R : S \to T$ is a partial relation from S onto T.
$S \xrightarrow[R]{+ \quad +} T$	$R : S \to T$ is a total relation from S onto T.

the relation is partial. By similar reasoning, if the *source* of the edge is labelled '1' or '+', this implies that the relation is *onto* its *destination*, because every element of the codomain is associated with at least one element of the domain. On the other hand, '?' or '$*$' implies that it is *into* its codomain. The rule — which is logical, if a little confusing — is to examine the *codomain* labelling to see if the relation is total or partial and to examine the *domain* labelling to see if it is *into* or *onto*. All sixteen possible labellings are explained in Table 2.11.

If the destination of an edge is labelled '1' or '?', this implies that the relation is a function, because no element of the domain maps to more than one element of the

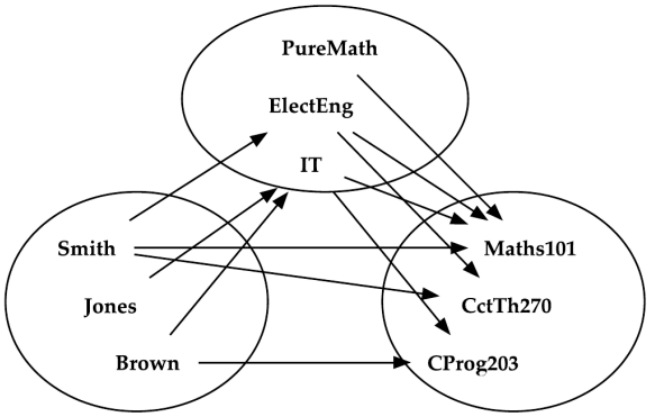

FIGURE 2.24

An *instance* of Figure 2.23. *Smith* is admitted to *ElectEng* and enrolled in *Maths101* and *CctTh270*. *Jones* is admitted to *IT* but is not currently enrolled in any subject. *Brown* is admitted to *IT* and is enrolled in *CProg203*. No one is currently admitted to *PureMath*. *PureMath* comprises *Maths101*, *ElectEng* comprises *CctTh270* and *Maths101*, and *IT* comprises *CProg203* and *Maths101*.

codomain. Conversely, if the destination is labelled '$*$' or '+', the relation is *not* a function.

Correspondences introduce a special case. An edge such as $S \xrightarrow[C]{?\quad 1} T$ means that C maps an element of S to exactly one element of T, but its converse maps an element of T to at most one element of S. In the case that C is an identity relation, this means that S is actually a subset of T. In such a case, we omit the name of the relation, and label the edge $S \xleftrightarrow{?\quad 1} T$. The edge is given two arrowheads.

Projection functions introduce a second special case. A **composite domain**, such as $X \times Y \times Z$, has nine total projection functions: into X, Y, Z, $X \times Y$, $X \times Z$, $Y \times Z$, $Y \times X$, $Z \times X$, and $Z \times Y$. Such projection functions are normally labelled π_X, π_Y, etc.

It is important to distinguish a schema from the graphs that it summarises. Vertices of the schema represent domains, i.e., *sets* of vertices. Edges of the schema represent *sets* of pairs in a relation. Since many graphs can map to the same schema, any particular graph that does so is called an **instance** of the schema.[20] Compare the instance of Figure 2.24 with the schema of Figure 2.23.

[20] This is the terminology used in database theory. In the jargon of artificial intelligence, such a graph is called a semantic network.

2.6 REPRESENTING SETS

Most readers will be familiar with the programming techniques described below; they are included here to reinforce the notion of 'set' in a programming context, thereby integrating many different data structures into a single concept.

In computer languages that support multiple assignments to variables,[21] a set is stored as a dynamic, or modifiable, object. Set representations should support several operations, including adding an element to a set, removing an existing member from a set, testing if a given element is a member of a set, enumerating the members of a set, finding the union, intersection, etc., of two sets, and testing if two sets overlap, are disjoint, etc. There is no universal best way to represent a set; the best choice depends, among other things, on which operations occur most often.

As a running example, we shall use the set obtained by adding the sequence of alphabetic character strings [**'first'**, **'second'**, **'third'**, **'fourth'**, **'fifth'**, **'sixth'**] to an empty set.

2.6.1 ARRAYS

A set S may be represented by an array, A, of length N, where $|S| = n \leq N$. Typically, entries $1 \ldots n$ contain representations of the elements of S, the remaining entries being unused; if $A(i) = x$, $(1 \leq i \leq n)$, then $x \in S$.[22] The entries of A may be ordered or unordered. In an **unordered array**, a new element is added at position $n + 1$ (n is then increased by 1). (See Figure 2.25.)

In an **ordered array**, the entries are stored in ascending (or, more rarely, descending) order of *value* (e.g., **fifth**, **first**, **fourth**, **second**, **sixth**, **third**). (See Figure 2.26.) The representation of the set is independent of the order in which the values were added to it. On the other hand, if a new element x is added, where $A(j-1) < x < A(j)$, entries $A(j) \ldots A(n)$ must be moved to higher locations so that x may be inserted at location $A(j)$. In Figure 2.26, the element **seventh** should occupy the 5th array entry, so **sixth** and **third** need to be moved up into the 6th and 7th entries.

Adding an element to an unordered array takes constant time, but adding an element to an ordered array takes time $O(|S|)$. On the other hand, it takes time $O(|S|)$ to test if $x \in S$ in an unordered array, but only time $O(\log |S|)$ in an ordered array, using **binary search**.[23] Similarly, the ordered representation allows the union,

[21] That is to say, languages such as *C*, *Java*, etc. In single-assignment languages, such as *Prolog* or *Haskell*, a *new* set must be formed, rather than an existing set being modified. This can make some of the techniques described here inefficient.

[22] Strictly, if we want to argue *formally* that data structure D represents set S, we first need to define a **representation function** Rep such that $S = Rep(D)$. A representation function needn't be a one-to-one correspondence. Many *unordered* arrays can represent a given set S. (We leave things informal here.)

[23] The middle element of the array is tested first; if this isn't the element that is sought, the search is repeated using either the smaller entries or the larger entries, as appropriate. To locate **second** in Figure 2.26, **fourth**, **sixth**, and **second** would be inspected, in that order.

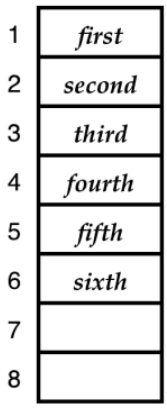

FIGURE 2.25

An Unordered Array. The element '**seventh**' would be added as its 7th entry.

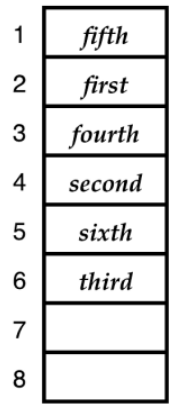

FIGURE 2.26

An Ordered Array. The element '**seventh**' would be added as its 5th entry, requiring the existing 5th and 6th entries to be moved.

intersection, etc., of sets S_1 and S_2 to be found in time $O(|S_1| + |S_2|)$ by merging,[24] whereas the unordered representation takes time $O(|S_1| \times |S_2|)$. Finally, enumerating the elements of an ordered array in ascending or descending order takes time $O(|S|)$, whereas it takes time $O(|S| \log |S|)$ for an unordered array.

Therefore, if many elements are stored but few retrieved, an unordered representation may prove better; if many elements are retrieved, the ordered representation may be better.[25] *This illustrates a general point about all data representations: the time spent in maintaining a higher degree of organisation may be more than recouped in other operations.*

The chief problem with array-based representations is that once N has been chosen, it cannot be increased.[26] Fortunately, several representations exist in which additional storage can be allocated dynamically.

2.6.2 LINKED LISTS

The simplest dynamic representation is a **linked list**. Such a list consists of nodes,[27] each of which is labelled with the value of a member of the set, and which has a directed branch to the remaining nodes of the list. The list must be accessed starting

[24] Pairs of entries from each array are compared, starting with the first ones. If both are equal, comparison continues with the next pair of entries from each array. If they are unequal, the lesser entry is processed, and the greater entry and the entry *following* the lesser entry form the next pair to be compared.

[25] It may sometimes be best to add elements to an unordered array, then sort the array into order.

[26] This isn't entirely true. It may be possible, for example, to dynamically create a larger array, and move the existing entries into it.

[27] We use the word 'node' because a list is technically a unary tree, whose root is the head of the list.

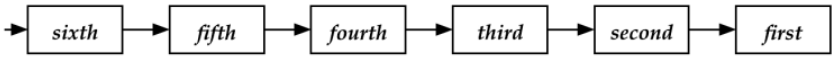

FIGURE 2.27

An Unordered Linked List. Elements are added to the head of the list in order of arrival. The element '**seventh**' would be added as its head.

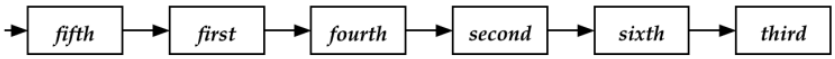

FIGURE 2.28

An Ordered Linked List. Elements are inserted in alphabetical order. The element '**seventh**' would be added between '**second**' and '**sixth**'.

with its first node, or **head**. The last node contains a special **null link**. When a new element is added to the set, a new node is created and added to the list.

As with arrays, the elements may be unordered (Figure 2.27), or ordered (Figure 2.28). When a node is added to an unordered list, it is most conveniently added at its head. Thus, the head of the list in Figure 2.27 is **sixth**, the most recent element to have been added. When a new node is added to an ordered list, it is added in the correct position, which merely needs two links to be adjusted.

As with arrays, the ordered representation allows the union, intersection, etc., of sets S_1 and S_2 to be found in time $O(|S_1|+|S_2|)$, whereas the unordered representation takes time $O(|S_1| \times |S_2|)$. Unfortunately, it takes time $O(|S|)$ to test if $x \in S$ *whether the list is ordered or not*; the nodes of a list can only be accessed in order from first to last. Search trees overcome this problem.

2.6.3 SEARCH TREES

A **binary search tree** is a rooted binary tree with labelled nodes. The nodes are labelled with representations of the members of the set. Each node except the root has an incoming branch from its parent. For uniformity, each node is considered to have two outgoing branches, called *left* and *right*. Each branch is either directed to a sub-tree or is **null**. (If both its branches are **null**, the node is a *leaf*.) In Figure 2.29 the root of the tree contains the value **first**. The nodes labelled **fourth**, **fifth**, and **sixth** are leaves. The set of elements with values less than **first**, {**fifth**}, is found by following the *left* branch, the set with greater values, {**fourth**, **second**, **sixth**, **third**}, by following the *right* branch. Such a tree is said to be ordered from left to right, and algorithms exist for listing its elements in ascending or descending order in time $O(|S|)$.

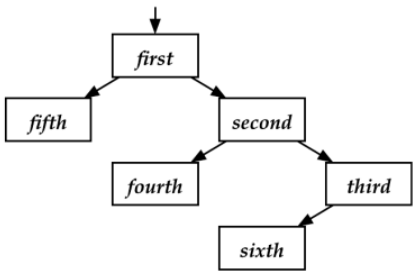

FIGURE 2.29

A Binary Search Tree. Elements are in order from left to right. Existing nodes aren't moved. The element '**seventh**' would be added to the left branch of '**sixth**'.

To test if an element $x \in S$, and therefore in the tree representing S, first the root is tested. If the node contains x, then $x \in S$. Otherwise, if x is less than the root, the *left* sub-tree is searched (recursively); if it is greater than the root, the *right* sub-tree is searched. If the sub-tree is **null**, then $x \notin S$.

When a new element x must be added to a set, in the simplest algorithm, the tree is first searched for x, until a **null** branch is found. A new leaf node labelled with x is then created, which replaces the **null** branch. To add **seventh** to the tree of Figure 2.29, the nodes **first**, **second**, **third**, and **sixth** would be inspected, and **seventh** would be added as the left branch of **sixth**. Unfortunately, if elements are added in certain sequences, the tree becomes tall and spindly. For example, the sequence, **fifth**, **first**, **fourth**, **second**, **sixth**, **third**, results in a tree like the ordered linked list of Figure 2.28.

If both sub-trees of every node in a binary tree have equal weight and height, the tree is said to be **perfectly balanced**. A perfectly balanced binary tree of height h contains $2^h - 1$ nodes. (See Figure 2.30.) Conversely, the height of a balanced binary tree is proportional to $\log_2 |S|$. For example, a balanced tree of height 3 contains 7 nodes; any member of the set it represents can be found by inspecting at most 3 nodes, i.e., in time $O(\log |S|)$. Searching is most efficient when a tree is perfectly balanced. Accordingly, search trees (and their sub-trees) are often **balanced** by rotating them, i.e., choosing a new root.

In practice, maintaining *perfect* balance proves too costly: additions to the tree can take time $O(|S|)$. Fortunately, several approximate balancing methods have been developed for which balancing only takes time $O(\log |S|)$. As a result, inserting a new element into such trees takes total time $O(\log |S|)$, and trees representing the union, intersection, etc., of sets S_1 and S_2 can be constructed in time $O(|S_1| + |S_2|)$.

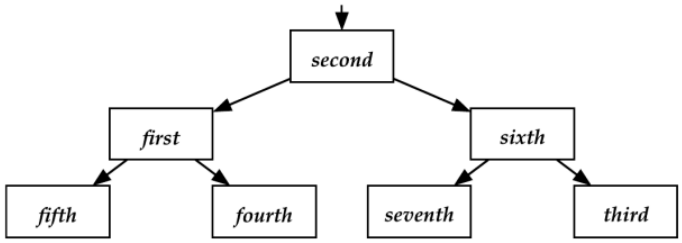

FIGURE 2.30

A perfectly balanced binary search tree.

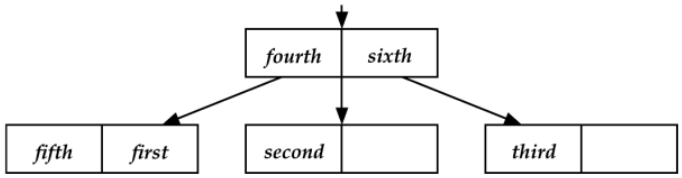

FIGURE 2.31

A B-tree with $b = 2$. The element '**seventh**' would be inserted into the empty cell adjacent to '**second**'.

2.6.4 B-TREES

It is possible to construct trees with orders higher than binary: in an n-ary tree, each node contains $(n - 1)$ elements and n branches to other nodes. For example, a ternary (3-ary) search tree has nodes containing 2 elements and 3 branches.

Of particular interest to us here are **B-trees**, which are widely used to implement large databases stored in secondary storage. For efficiency, information is transferred between secondary storage and RAM in blocks. Typically, the nodes of the tree correspond to blocks and contain many elements and links.

A B-tree is balanced by controlling the number of elements in each node. Each node of a B-tree except the root contains at most b and at least $b/2$ elements, and at most $b + 1$ links. If a node contains b elements, it is said to be **full**. If an attempt is made to add an element to a full node, it is said to **overflow**, and a new node is allocated to the tree.[28] Conversely, if removing an existing member of the set would reduce the number of elements in a node below $b/2$, a node may be removed.[29] Figure 2.31 illustrates the simplest possible B-tree, with $b = 2$ and maximum degree 3.

[28] This may cause the parent node to overflow, and so on. If the root overflows, a new parent root is created, and the height of the tree increases by 1.

[29] Which may cause the parent to underflow, and so on, possibly reducing the height of the tree by 1.

B-trees always have uniform height: every leaf is the same distance from the root. In the best case, when all the nodes are full, a tree of height h represents a set with $|S| \approx b^h$ (for large b). In the worst case, when all the nodes are half-full, it represents a set with only about $(b/2)^h$ members.[30] Compared to binary trees, B-trees are shallow. For example, a B-tree with $b = 64$ and $h = 2$ can store as many as 4,224 $(64 + 65 \times 64)$ elements, but a comparable perfect binary tree has $h = 12$. Since a search for a particular element must inspect all nodes along its path from the root, the height of the tree determines the number of blocks that need to be read from secondary storage.

2.6.5 HASH TABLES

All the representations we have discussed so far may be considered to be trees of various degrees. (An array is a tree of height 1 and degree $|S|$, a linked list is a tree of height $|S|$ and degree 1.) One representation stands apart from the rest: the hash table.

A **hash function** is any function h that maps the values of the elements of a set onto the range $1 - p$. A typical hash function is to regard the internal representation of x (regardless of its type) as a binary number X, and choose $h(x) = (X \bmod p) + 1$, where p is a prime number roughly equal to $|S|$. The effect is to partition the elements of the set into p subsets. Each element x is a member of subset $h(x)$.

The subsets are located using an array called a **hash table**. Given an element x, its corresponding subset is found via entry $h(x)$ of the array. The average size of a subset is $|S|/p$. If $p \simeq |S|$, the subsets will contain an *average* of about 1 element, and take little time to search. Since it also takes little time to compute a hash function, hashing can greatly speed access to large sets. Sadly, in the (unlikely) worst case, all the elements might be assigned to the same subset, and all the other subsets might be empty. If no subset contains more than one element, the hash function is said to be **perfect**. The subsets themselves may be represented in any of the ways we have discussed above (including structures that utilise unused entries within the hash table). When the subsets are represented in a way that takes time $O(|S|)$ to access (e.g., a linked list), hashing *divides* the access time by a factor of p. On the other hand, if the subsets can be accessed in time $O(\log |S|)$ (e.g., a tree), hashing merely *subtracts* a constant time $O(\log p)$.

Although hash tables speed up the search for a given element, they make it less easy to enumerate the elements of a set in ascending or descending order.

2.6.6 BIT MAPS

An array of n bits (a **bit map**) may be used to represent a finite set of integers. Assume bits $1 \ldots n$ represent the integers $1 \ldots n$. If bit n is **on** (1), n is a member of the set; if bit n is **off** (0), n is not a member of the set. (By extension, a bit map can represent any collection of elements that corresponds to a finite range of

[30] Various refinements of the basic idea are used to avoid the worst case.

integers.) Such a bit map can occupy one or more words in memory. For example, a computer with 64-bit words can represent the integers 1–128 using two words. This representation is actually a *function* from each element to a Boolean value, and is called the **characteristic function** of the set.

To add an element to the set, it is merely necessary to turn the corresponding bit on; to remove it, the bit is turned off. The union of two sets is found using logical **or**, their intersection by **and**, and so on. Most computers have built-in operations that can manipulate bit maps in these ways.

2.6.7 DECISION TREES AND DECISION DIAGRAMS

A **decision tree** is a rooted tree with labelled *edges*, in which a path from the root to a node *spells* the value of an element. By 'spells', it is meant that its value is given by the *sequence* of the labels on the edges along the path.

A **binary decision tree** has edges labelled 0 or 1. To add a new element, its representation is considered as a binary number, and nodes are added as necessary to create a path that spells the number.

An **alphabetic decision tree** (or **dictionary tree**) has edges labelled by alphabetic characters, so that paths literally spell character strings. To represent a set of strings of unequal length, a path may need to terminate at an internal node, which must therefore be marked to indicate that it ends a possible path.[31]

A **decision diagram** is an optimised form of decision tree. If two sub-trees anywhere within the tree are identical, only one instance is stored. Both parent nodes then branch to the remaining instance.[32] The resulting directed acyclic graph is *not* a tree. Because identical sub-trees are merged, decision diagrams can be a very compact way of representing sets. Unfortunately, finding identical pairs of sub-trees is costly.

2.6.8 OTHER STRUCTURES

Many other data structures can represent sets. Some of them are mixtures of those we have described, for example, a linked list of arrays and an array of linked lists are both possible structures. Certain special representations are suited to particular problems. For example, a **union-find tree** has its branches directed from children to parents rather than from parents to children. It makes it easy to merge sets and to find of which set a given element is a member, but makes it less easy to enumerate the members of any given set.

---

[31] If, for example, the tree contained both the word 'boring' and the word 'boringly', they would share the same path, but the 'g' and 'y' in 'b-o-r-i-n-g-l-y' would both be marked to show that they end words.

[32] Consider, for example, that many English words share common endings. In an alphabetic decision diagram, the words 'boring' and 'exciting' would both share the sub-tree for 'ing'.

2.7 REPRESENTING FUNCTIONS, RELATIONS, AND GRAPHS

2.7.1 FUNCTIONS

A function from X to Y associates one y value with each x value. A simple set of $x \mapsto y$ pairs is an adequate way to represent a function, there being at most one pair for any given value of x. Therefore any structure that can represent the set X can be extended to represent a function. For example, we might choose to use a binary search tree whose nodes contain $x \mapsto y$ pairs to represent a function from X to Y. In addition, when X is, or can be mapped onto, a small range of integers, a function from X to Y can conveniently be represented by an array in which element x contains the corresponding value of y.

In practice, a system often needs to represent several functions with a common domain X. Suppose we have $f : X \to Y$ and $g : X \to Z$. Then it will usually prove wise to combine them into a single function $f \| g : X \to Y \times Z$, by storing the set of triples $x \mapsto (y, z)$. This has the advantage that the domain X is stored only once, saving space, and also saving time whenever $f(x)$ and $g(x)$ are both needed.

A small complication arises when f or g are *partial* functions. There may be some values of x for which there is a corresponding value of y but no corresponding value of z, or *vice versa*. In this case, some means must be used to show when there is no y (or z) value; y (or z) is said to be **null**. This can be implemented by reserving a special value of y (or z) that is not a true member of Y (or Z) or by storing additional bits to indicate which values are present.

2.7.2 CORRESPONDENCES

A correspondence is a function whose inverse is also a function. A correspondence $f : X \to Y$ is a set of $x \mapsto y$ pairs such that, not only is each x value unique, but also each y value is unique. This causes a difficulty: an ordered data structure of $x \mapsto y$ pairs, such as a search tree, offers a fast way to check if a new $x \mapsto y$ pair duplicates an existing x value, but no fast way to check if the y value is duplicated. The usual solution to this problem is to store the set of y values redundantly in a second data structure. In many applications, it may even pay to store the set of $y \mapsto x$ pairs, forming the inverse function f^{-1}.

If there are other functions of X, say $g : X \to Z$, then, as just mentioned, a single structure can be used, holding triples of the form $x \mapsto (y, z)$. If f is a correspondence, it is implicit that a second function $h : Y \to Z = f^{-1}\,;g$ must also exist, but it is pointless to store triples of the form $y \mapsto (x, z)$, as h can be found from f^{-1} and g by composition.

In the case that the correspondence is a subset relation, e.g., $Y \subseteq X$, then it is sufficient to store a function $X \to \{True, False\}$, where the pair $x \mapsto True$ indicates $x \in Y$, but the pair $x \mapsto False$ indicates $x \notin Y$.

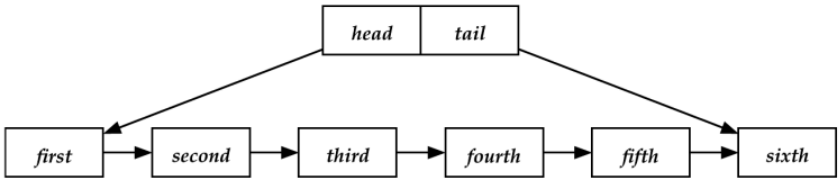

FIGURE 2.32

A linked list with pointers to both its head and its tail. Such lists can be concatenated in constant time.

2.7.3 SEQUENCES

A sequence of length n is, strictly speaking, a special case of a function whose domain is $1 - n$. In practice, certain operations commonly performed on sequences mean that storing the domain values explicitly would be a poor choice. For example, adding a new element to the start of a sequence means that all the existing elements must be renumbered. Likewise, concatenating a second sequence onto the end of a first (e.g., concatenating $[a,b,c]$ and $[d,e,f]$ to give $[a,b,c,d,e,f]$) means that the elements of the second sequence must be renumbered. To avoid renumbering, the numbering should be implicit. This is easy in the case of a linked list; the first element is implicitly numbered 1, the second numbered 2, and so on. To concatenate two lists, the last node of the first list simply has to be redirected to point to the head of the second list. To speed this process, a pointer may be maintained to point directly to the last element of the list. (See Figure 2.32.)

A common use of such a linked list is to represent a first-in, first-out (FIFO) **queue**, in which elements are added one-by-one to the **tail** of the queue and removed one-by-one from the head. A similar list structure is used for a last-in, first-out (LIFO) queue or **stack**, where elements are both added to and removed from the head. In this case, the tail pointer is unnecessary.

The problem with a linked list is that it doesn't offer a fast way of accessing its ith member. For very long sequences, a better way is to use a binary tree. Each node of the tree contains a member of the list, pointers to sub-trees containing the elements that precede it and follow it, and the weight of its sub-tree. (In the case of the root, its weight equals the length of the sequence.) The position of any node in the sequence is one plus the weight of its left sub-tree. Typically the tree is approximately balanced by weight. Any element can be found in time $\mathbf{O}(\log n)$. Also, adding or deleting an element requires only $\mathbf{O}(\log n)$ nodes to be rebalanced.

A **heap** is a special kind of balanced binary tree used to represent **priority queues**. A priority queue is a sequence in which nodes are ordered by priority — earliest priority first. The root of any sub-tree has an earlier priority than any of its descendants, so the root (top) of the heap has the earliest priority of all. The tree is filled strictly from top to bottom, left to right. A node added to the tree starts at the

bottom right and is then made to sift up to its correct position. When the earliest priority node is removed from the root, it is first replaced by the bottom right node, which is then made to sift down to its proper level. Sifting up and sifting down both take time $\mathbf{O}(\log n)$.

2.7.4 RELATIONS AND GRAPHS

All the set representations we described earlier may be adapted to represent relations. There are two ways to proceed: we either represent the set of pairs in the relation, or we associate each value in the domain with its corresponding *set* of values in the codomain, called its **image**.

In a linked list consisting of $x \mapsto y$ pairs, it takes time $O(|S|)$ to test for the presence of a given value of $x \mapsto y$ or to enumerate all pairs with a given value of x or a given value of y. In contrast, an ordered array allows a given value of $x \mapsto y$ to be found in time $O(\log |S|)$ using binary search, and, since all pairs with a given value of x will be stored adjacently, binary search can be adapted to allow all pairs with a given x value to be enumerated efficiently. Unfortunately, a structure that is ordered by $x \mapsto y$ is *not* ordered by $y \mapsto x$, so the whole array must be scanned to enumerate all pairs with a given value of y. Other ordered structures, such as search trees, or decision diagrams, have a similar behaviour.

The second approach stores pairs of the form $x \mapsto \{y_1, y_2, \dots\}$.[33] It has the advantage that it can better represent partial relations. A pair of the form $x \mapsto \{\}$ means that x has no corresponding y value. Such a representation is usually called a **graph data structure**, where the x values correspond to vertices and the y values correspond to out-edges. Any of the set representations may be used to represent the set of vertices, and any representation (not necessarily the same) may be used to represent the sets of edges. The structure is easily adapted to represent a labelled graph.

There are three commonly used structures of this type.

2.7.4.1 *Adjacency Matrices*

The first structure is a two-dimensional array, or **adjacency matrix**.[34] The rows of the matrix represent x values, and the columns represent y values. The entry in cell (x, y) is *true* if $x \mapsto y$ is a member of the relation, and is *false* otherwise. By extension, a matrix can also represent a labelled graph, the entry in cell (x, y) containing the label of the edge $x \to y$. The matrix representation allows all (x, y) pairs with a given x value or a given y value to be easily enumerated (by scanning row x or column y), but it usually makes poor use of memory: the number of cells is $X \times Y$, which may be many times more than the number of pairs in the relation.

[33] Strictly, this is not a relation from X to Y, but a *function* from X to 2^Y, the set of all subsets of Y.
[34] We described adjacency matrices in Section 2.5.5.

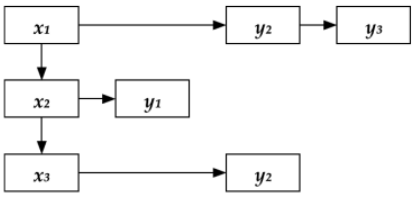

FIGURE 2.33

An adjacency list. The domain contains the values x_1, x_2, and x_3; the codomain contains y_1, y_2, and y_3.

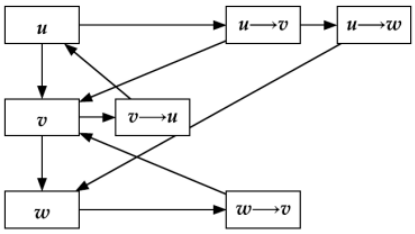

FIGURE 2.34

A graph data structure. Paths in the graph can be traced following the links between cells.

2.7.4.2 *Adjacency Lists*

The second structure is called an **adjacency list**. (See Figure 2.33.) It consists of a list (or possibly an array) of domain values, each cell of which has a list of out-edges labelled with values in the codomain.

In the case of a homogeneous relation, where the domain and codomain are the same, each edge cell is often a pointer to a vertex, as in Figure 2.34. This graph data structure is the basis of some very fast algorithms, based on a procedure known as **depth-first search** (**DFS**).

In a depth-first search of a *vertex*, the vertex is visited, all its successors are visited and searched *recursively*, and then the vertex is visited again. The first visit to the vertex is called its **preorder** visit, and the second visit is called its **postorder** visit. Any vertex that has already been visited during the search is ignored, which prevents an infinite recursion in the case of a cycle. The recursion means that *all* the vertex's successors are visited before its postorder visit.

For example, a DFS of u in Figure 2.34 would make a preorder visit to u, then follow the edge $u \rightarrow v$ to v. Continuing recursively, DFS would then follow the edge $v \rightarrow u$, but finding that u had already been visited, the recursion would unwind, and v would be visited in postorder. The DFS would continue to unwind to u, then follow its second edge to w. Since w would not yet have been visited, the edge $w \rightarrow v$ would then be considered, but since v had been visited, the recursion would unwind twice, making a postorder visit to w, and finally a postorder visit to u.

A depth-first search of a *graph*, shown in Algorithm 2.4, is a DFS of all its (as yet unvisited) vertices. A DFS of the graph of Figure 2.34 would begin with a DFS of vertex u, then consider v and w, but they would not become the starts of new searches because they were already visited during the search from u. A DFS of a graph visits each vertex and each edge exactly once and has complexity $O(|V| + |E|)$, where $|V|$ and $|E|$ are the numbers of vertices and edges of the graph.[35]

[35] $|E|$ is at most $|V|^2$.

Algorithm 2.4 Depth-first search of a graph.

1. Initialise the set *Visited* to be empty.
2. For each vertex V of the graph,
   - If V is not a member of *Visited*
     a. Add V to *Visited*.
     b. Visit V in preorder.
     c. For each successor U of V, make a depth-first search of U recursively, as in Step 2.
     d. Visit V in postorder.

Depth-first search has the useful property that the postorder visits to the vertices of an acyclic graph occur in reverse topological order.[36] In the case of a cyclic graph, the postorder visit to the first vertex visited in a strongly connected component doesn't occur until every other vertex in the component has been visited. Depth-first search is therefore the basis of algorithms that can find the transitive closure or the strongly connected components of a graph in time $O(|V| + |E|)$.

The graph data structure doesn't offer any convenient way of finding the set of *in*-edges of a vertex, although there is a fast algorithm for reversing *all* the edges to find the converse relation.

2.7.4.3 *Sparse Matrices*

The third structure is an elaboration of the adjacency list: the **sparse matrix** representation. Imagine an adjacency list representation of a relation and another adjacency list representation of its converse. Now share both sets of edge cells, as in Figure 2.35.

2.7.5 FILES

A **file** is a data structure stored on a persistent secondary storage medium. Physically, the medium is arranged in blocks of fixed size, e.g., 4 KB. The computer's operating system will usually try to arrange that a file occupies a contiguous series of blocks, or if that isn't possible, a few contiguous areas, called **extents**.

From the programmer's point of view, a file consists of **records**. There are typically several records per block. Usually, the operating system unpacks the blocks into individual records when they are read and packs records into blocks when they are written. Each record typically represents one element of a set, one pair of a relation, or one pair, triple, etc., of one or more functions. The components of records are called **attributes**. For example, a function from X to Y would have attributes named X and Y.

[36] Because their successors are first visited recursively.

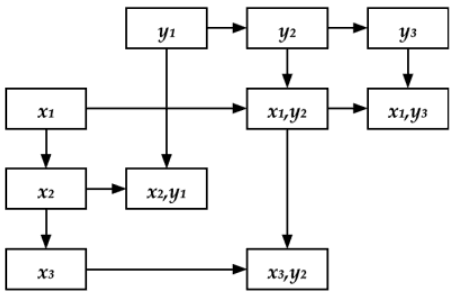

FIGURE 2.35

A sparse matrix. Each edge cell has two links to other edge cells. In mathematical applications involving matrices, the domain and codomain arrays replace the lists, and only non-zero values are explicitly represented.

As a rule, each record has a unique identifier: each pair in a *function* from X to Y must have a different value from the domain X. X is called a **key** of the file; Y is a **non-key attribute**. In the case of a *relation* from X to Y, only the $x \mapsto y$ *pairs* are unique, so the file has the **composite key** (X,Y). In the case of a sequence, records are physically stored in order of the sequence.

Records don't have to be fixed in length. For example, a file could represent a graph, in which each record could contain a vertex followed by a list of edges of arbitrary length.

A file can represent any data structure. Records can be added to a file dynamically, so that dynamic data structures can be created.

A record has both an absolute and a relative address. The **absolute address** specifies the physical storage location (e.g., surface, track, sector and byte) of the record. The **relative address** specifies the number of bytes from the start of the file. The computer's operating system translates relative addresses into physical addresses by reference to the file's directory entry, which lists the physical locations of its extents. When a file is copied, all its physical addresses change, but the relative addresses stay the same. Therefore, records can be linked into list or tree structures using their relative addresses.

Most modern computer systems allow an entire file to be read into and processed entirely within RAM, the file being saved periodically and, again, when processing is complete. The file can therefore contain whatever data structures are convenient to the application. There is one restriction: it cannot be guaranteed that a saved file will occupy the same memory locations when it is later reloaded into RAM. Therefore, the file cannot contain absolute memory addresses. It may, on the other hand, contain relative addresses (i.e., relative to the start of the file), or it may be saved in a form that is free of addresses. For example, it can be reduced to a textual form with mark-up, such as *XML*.

On the other hand, when the volume of data exceeds the available RAM, or when saving must be continuous, it is wise to deal with the persistent data directly. In such a case, each file usually represents one set, one relation, a cluster of functions, or a sequence.

For our purposes, two kinds of files are important.

A **sequential file** consists of a sequence of records stored one after another. A sequential file is read or written sequentially, starting with the first record, then the second record, and so on.

An **indexed-sequential file** is indexed, allowing the records to be located either by key or in sequential order. Records may be added or deleted anywhere within the file. The file is structured to make sequential access efficient: blocks contain records with contiguous key values, but the blocks may only be partly filled, and the blocks themselves may not always be stored in key order. A B-tree structure is often used, with the nodes of the tree mapped directly onto blocks. Often, it is possible to store the entire index of a file in RAM, so that, after consulting the index, a record can be located by direct access to the block that contains it.

An indexed file can have several indices, allowing access via more than one attribute. It is also possible for several records to share the same value of a secondary key, in which case the index will need to store the *set* of records with each key value, perhaps as a linked list.

Finally, it is perhaps worth mentioning that the directories or folders that a hierarchical file store uses to contain files form a kind of decision tree, in which the directory names and filename spell the path from the root to the file. For example, the path '/Users/barry/Documents/Synthesis/Maths.tex' describes a path from the root of the file system to the document I am currently editing. Each step in the path translates a name into the storage address of the next node.

2.8 SUMMARY

It should be obvious to the reader that there are strong connections between the various ideas presented in this chapter. *Relations* are just special kinds of *sets*. Although some relations are best defined by enumeration (such as who is the parent of whom), others are better defined by *predicate calculus* expressions. *Graphs* are a way of representing relations that is better suited to the human visual system than matrices or lists of pairs. *Labelled graphs* are often a superimposition of several simpler graphs.

For example, if $R = \{x \mapsto y \mid P(x, y)\}$, the ideas of relation, graph, and predicate are connected as follows:

$$x\ R\ y \Leftrightarrow x \underset{R}{\mapsto} y \Leftrightarrow P(x, y)$$

Finally, *schemas* are special graphs that describe relations between relations.

Table 2.12 on the next page lists the mathematical symbols explained in this chapter.

Table 2.12 The mathematical symbols explained in this chapter

Symbol	Name	Meaning
$\neg$	not	$\neg P$ is false if P is true and true if P is false.
$\wedge$	and	$P \wedge Q$ is true only if both P and Q are true.
$\vee$	or	$P \vee Q$ is false only if both P and Q are false.
$\Rightarrow$	implies	$P \Rightarrow Q$ is true unless P is true and Q is false.
$\Leftarrow$	if	$P \Leftarrow Q$ is true unless P is false and Q is true.
$\Leftrightarrow$	if and only if	$P \Leftrightarrow Q$ is true when P and Q have the same value.
$\forall$	for all	$\forall x(P(x))$ is true only if $P(x)$ is true for all x.
$\exists$	there exists	$\exists x(P(x))$ is false only if $P(x)$ is false for all x.
$\in$	is a member of	$X \in S$ is true only if x is a member of set S.
$\notin$	is not a member of	$X \notin S$ is true only if x is not a member of set S.
$\mid$	such that	$\{X \mid P\}$ is the set of all X such that P is true.
$\mid\ \mid$	cardinality	$\mid S \mid$ denotes the number of elements in S.
$\mathbf{U}$	universe of discourse	The set of all elements x under discussion.
$\emptyset$	empty set	The empty set, alternatively written as $\{\}$.
$\subseteq$	subset	$S \subseteq T$ if and only if $x \in S \Rightarrow x \in T$.
$\subset$	proper subset	$S \subset T$ if and only if $S \subseteq T \wedge S \neq T$.
$\cup$	union	$x \in S \cup T$ if and only if $x \in S \vee x \in T$.
$\cap$	intersection	$x \in S \cap T$ if and only if $x \in S \wedge x \in T$.
$\setminus$	set minus	$x \in S \setminus T$ if and only if $x \in S \wedge x \notin T$.
$\times$	Cartesian product	$(x, y) \in S \times T$ if and only if $x \in S \wedge y \in T$.
$\underset{R}{\mapsto}$	maps to under R	$x \underset{R}{\mapsto} y$ if $x\ R\ y$. (Alternatively, $x \underset{R}{\mapsto} y$ if $(x, y) \in R$.)
$;$	relational product	$x\ R\ ;S\ z$ if and only if $\exists y(x\ R\ y \wedge y\ S\ z)$.
$\|\|$	parallel composition	$x\ R \| \| S\ (x, y)$ if and only if $x\ R\ y \wedge x\ S\ z$.
R^n	relational power	$R^n = R\ ;R\ ;R \ldots$, where R occurs n times.
R^{-1}	relational converse	$x\ R^{-1}\ y$ if and only if $y\ R\ x$.
R^0	relational identity	The identity function on R; $x\ R^0\ x$ if $\exists y(x\ R\ y)$.
R^+	transitive closure	$R^+ = R \cup R^2 \cup R^3 \cdots$.
R^*	reflexive transitive closure	$R^* = R^0 \cup R^+$.
$u \underset{G}{\rightarrow} v$	labelled directed edge	There is an edge from u to v in G.
$u \rightarrow v$	directed edge	$u \underset{G}{\rightarrow} v$, where G is understood.
$u \rightarrow^+ v$	directed compound path	There is a directed path of length ≥ 1 from u to v.
$u \rightarrow^* v$	directed path	There is a directed path of length ≥ 0 from u to v.
$u \leftrightarrow v$	undirected edge	$u \rightarrow v \wedge v \rightarrow u$.
$u \leftrightarrow^+ v$	undirected compound path	$u \rightarrow^+ v \wedge v \rightarrow^+ u$.
$u \leftrightarrow^* v$	undirected path	$u \rightarrow^* v \wedge v \rightarrow^* u$.

Provided we use graphs formally, as representations of relations, being clear what the underlying domains and relations are, we shall keep out of trouble. There are two dangers we must avoid: being sloppy about what relations are being represented, or piling so much information onto a single graph that our visual systems can no longer make sense of it.

We reluctantly admit that graphs are mainly useful for tutorial purposes, because real-world relations are typically so complex that their graphs overwhelm the eye. They are also time-consuming to draw, especially if care is taken to minimise the crossing of edges. In practice, a list-of-pairs representation or a matrix representation is often better.

We observe that there are many data structures that can be used to represent sets and relations within a computer system, both within RAM and in secondary storage. This chapter has only listed the most common possibilities. As a general rule, the more time is devoted to storing an element, the less is spent in retrieving it. Structures and algorithms exist that are able both to store and to retrieve the elements of a set S in time $O(\log |S|)$.

Conversely, all the various data structures that programmers have devised can be seen as realisations of a few mathematical concepts. It is better to work first at a conceptual level, and defer implementation decisions by naming functions and procedures abstractly — as in $Member(x,S)$, rather than $SearchTree.Find(x,S)$. Whether to use a search tree or an ordered list, etc., can be decided later.

A theme that will occur several times in the following chapters is the usefulness of finding the strongly connected components of directed graphs. Consider the graph of Figure 2.14, and imagine that it was derived from the following set of six simultaneous equations:

$$\begin{aligned} a &= e - a - 12 \\ b &= a + 2c - d \\ c &= 3a + 1 \\ d &= 2c - e + 3f - 7 \\ e &= 2c + 2 \\ f &= 2a - 3b + e \end{aligned}$$

where there is an edge in Figure 2.14 from variable x to variable y if x is on the right-hand side of the equation for y.

In Figure 2.18, we saw that this graph has two strongly connected components, $\{a,c,e\}$ and $\{b,d,f\}$. The implication is that we have two separable sub-problems, one for each component:

The set of three equations

$$\begin{aligned} a &= e - a - 12 \\ c &= 3a + 1 \\ e &= 2c + 2 \end{aligned}$$

is independent of b, d, and f and has the solution $a = 2$, $c = 7$, $e = 16$. We can then substitute these values into the remaining equations to obtain

$$b = 2 + 14 - d$$
$$d = 14 - 16 + 3f - 7$$
$$f = 4 - 3b + 16$$

Thus, we have two small problems rather than one big one.

In general, finding the strongly connected components of a graph breaks it into more manageable subgraphs. Since the resulting reduced graph is acyclic, it has a topological sort, which lets each sub-problem be tackled in turn. Unfortunately, the cycles *within* a sub-problem may make it hard to understand or hard to solve. In the following chapters, we shall see over and again that it is cyclic relationships that cause problems for the systems analyst and the implementor.

2.9 FURTHER READING

I have yet to find a single textbook that includes all the material in this chapter, although chapters 5–9, 12, and 14 of 'Foundations of Computer Science: C Edition' by Alfred V. Aho and Jeffrey D. Ullman (1994, ISBN: 0-7167-8284-7) come close.

Most of the mathematical ideas can be found in Kenneth A. Ross and Charles R. Wright, 'Discrete Mathematics' (ISBN: 0-13-065247-4), but it doesn't cover some of the material on graphs and relations. This material is covered in 'Relations and Graphs' by Gunther Schmidt and Thomas Ströhlein (ISBN: 3642779700), but in far more depth than is needed in this book. Representations of sets, relations, and graphs can be found in 'Data Structures and Algorithms' by Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft (ISBN: 0-201-00023-7). There are many other good books that contain similar material, but before borrowing one, check that its index contains a reference to 'strongly connected components of a graph'.

Warshall's Algorithm for finding the transitive closure of a relation was published in January 1962 under the rather obscure title, Warshall, Stephen, 'A theorem on Boolean matrices', *Journal of the ACM* 9 (1): 11–12. The article by James Eve and Reino Kurki-Suonio, 'On computing the transitive closure of a relation', *Acta Informatica*, 8(4), 303–314 (1977), describes a much faster graphical method, based on Robert Endre Tarjan's 1972 article 'Depth-first search and linear graph algorithms', *SIAM Journal on Computing* 1 (2): 146–160.

Finally, the paper, 'The transitive reduction of a directed graph', *SIAM Journal of Computing*, 1 (2): 131–137, (1972) by Alfred V. Aho, Michael Garey, and Jeffrey D. Ullman, describes an efficient way to compute a transitive reduction.[37]

[37] Yes, both these final two articles appeared in the same issue.

2.10 EXERCISES

1. Complete the proof of Equation 2.1.27.
2. Prove Equation 2.1.11. (Hint: Use Axiom 2.1.7 and the resolution theorem (page 29). Set out your proof as a vertical stack of propositions.)
3. Use Axiom 2.1.7 to reduce $P \Rightarrow Q$ to CNF, then use De Morgan's Theorem to find its logical complement. How do you interpret your result?
4. Why does a set of cardinality n have 2^n possible subsets? (See 2.3.1.)
5. In what special circumstances does $R^1 ; R^{-1} = R^0$? (See inequality 2.4.26.) (Hint: Consider the difference between the *Child* relation in connection with family relationships, and the *Child* relation in rooted trees such as Figure 2.19.)
6. By analogy with Equation 2.4.22, define the *Has Uncle* relation.
7. Represent the labelled graph of Figure 2.21 as a matrix, with entries '0', '1', '0,1' or blank.
8. Sketch a graph data structure corresponding to Figure 2.21 in the style of Figure 2.34, labelling its edges '0' or '1'.
9. Graph G has vertices V, W, X, Y, and Z, and edges $V \rightarrow W$, $V \rightarrow X$, $W \rightarrow V$, $W \rightarrow X$, $X \rightarrow Y$, $X \rightarrow Z$, $X \rightarrow Y$, $Y \rightarrow Z$, and $Z \rightarrow Y$. Find the strongly connected components of G and sketch its reduced graph. (See Section 2.5.5.)

CHAPTER

Atoms

3

CHAPTER CONTENTS

INTRODUCTION

We normally regard an information system as a model, or analogue, of part of the real world. The real world contains things, usually many things, that we wish to model. We shall call the real-world things we model **atoms**. We use the word 'atom' because they are the smallest things that we model. Depending on the nature of the system, the number of atoms to be modelled will vary. A flight control system has to model only one aircraft, with two wings, four engines, and so on, but a bank has to model thousands of customers.

We don't just want to model the existence of atoms, we also want to model their behaviour.

Of course, the analyst must choose what things in the real world need to be modelled by atoms. Further, the analyst should not, and cannot, model the chosen

Systems Analysis and Synthesis: Bridging Computer Science and Information Technology. DOI: 10.1016/B978-0-12-805304-1.00012-6

atoms in their entirety, but only certain aspects of them. For example, a bank is not much interested in what its customers do at home, but only in the transactions that they conduct with the bank. The flight control system of an aircraft is concerned with the position and velocity of the aircraft, but not with which passengers have asked for a cup of coffee.

The things that happen in the real world that cause us to inspect or update our model, we shall call **events**. **Update events** cause the **states** of atoms to change. Update events are **atomic**, or indivisible; an atom has a state *before* an event and a state *after* an event, but its state *during* an event is undefined. The states of atoms between updates, we shall call **snapshots**. Snapshots are represented by records in files, rows in database tables, or other data structures. Update events invoke **event procedures** (or methods) that transform a **before snapshot** into an **after snapshot**.[1] **Inspection events** simply analyse and report the states of atoms. They leave the states of atoms unaltered. Even so, it isn't unusual for events both to inspect and to update atoms, perhaps reporting the before snapshot or the after snapshot. When it is necessary to keep a permanent record of *events*, it is typically stored in sequential files.

Events of interest to a banking system might be the opening of a new account, depositing money, withdrawing money, a balance enquiry, printing a statement, and so on. We only need to inspect or update the snapshot of the account when these events happen in the real world. On the other hand, an aircraft control system must monitor the state of an airframe *continuously*. In a digital control system, this is approximated by updating the state at frequent intervals. The passage of time in the real world generates a series of internal clock tick events. The succession of resulting snapshots is like a movie.[2]

What we consider to be an event depends on our perspective. A person applying for a job might consider that completing a computer course was an event worth including in a curriculum vitae. On the other hand, the institution that offered the course would not only consider each subject completed as one event, but even more detail: enrolling, sitting an examination, etc. In this way, the course and the subject enrolments within it, although events at one level, become atoms at a greater level of detail. Think of a bank account: it is not enough for the bank to keep track of the balance of an account. A customer needs to see the history of transactions that have led to that balance. For that reason, records of deposits and withdrawals need to be kept, which need to be reported. In the context of updating a balance, a deposit is an event. In the context of being reported, the deposit is an atom.

The state of an aircraft's airframe may be described by 12 variables: its position relative to the earth is given by its latitude, longitude, and altitude. Each of these variables has a corresponding linear velocity. Its attitude is given by three angles: roll, pitch, and yaw. Each of these has a corresponding angular velocity. All 12 variables are *continuous* quantities that can be represented by real numbers, so the possible states of an airframe are therefore infinite in number.

---

[1] An object (in the sense used in object-oriented programming) might therefore correspond to an atom and some events that affect it.

[2] We shan't discuss continuous-time controllers in this book, only sampled-data systems.

The state of a customer's bank account is given primarily by the balance of the money owed to the customer. This can be described by an integer. If a customer may deposit any amount, without limit, the states of an account are *discrete*, but are also effectively infinite in number.[3]

We begin by considering discrete finite-state systems.[4] Our motive is that, by identifying the states of atoms, we discover what kinds of events a system needs to consider. Conversely, learning about the existence of a certain kind of event may help us to identify the need for a new state. This interaction between events and states can be very useful in eliciting information from a **domain expert**, a person familiar with the real-world situation that the system is meant to model. Indeed, as we shall learn, such analysis can help us discover additional atoms that we need to model.

Finite-state systems can even be used to model certain aspects of continuous or infinite-state systems. For example, a *Bank Account* might be considered to have one of three states: *Null*, *Open*, or *Closed*; and from the point of view of an airport, an *Aircraft* might be in one of four states: *Arriving*, *Landed*, *Boarding*, or *Departed*.

Our finite-state models will have only a few states: *we are concerned with states only in so far as they determine what events are possible*. When an *Aircraft* has *Departed*, it may climb, dive, bank, or yaw; when it has *Landed*, diving isn't really an option. When a *Bank Account* is *Open*, it may be credited or debited by any reasonable amount. When it is *Closed*, it may not.

When an aircraft has departed, it may have many possible spatial positions. When a bank account is open, it may have any reasonable balance. We therefore consider *Departed* or *Open* as **families of states**, which are related in that they allow the same set of events.

3.1 FINITE-STATE AUTOMATA

Systems that have a finite number of states are called **finite-state automata** (**FSA**s). We can model them by labelled directed graphs, called **state-transition diagrams** (see Figure 3.1). Vertices are labelled with the names of states, and edges are labelled with **transitions**: changes of state. In the present context, transitions are caused by events.

The vertex containing a dot represents the **initial state** of a finite-state automaton. This is the state of the automaton before any transitions have occurred. A vertex with a bold outline marks a **final state**. An automaton in a final state has done its job and may often cease to be of interest.

[3] An alternative view, discussed below, is that being in the customer's account is one of a finite number of states of some coins.

[4] Whenever a system is modelled by a digital computer, there is a trivial sense in which it *must* be finite-state, because the computer's memory has only a finite number of states. One bit has 2 possible states, one byte has $2^8 = 256$ possible states, and so on.

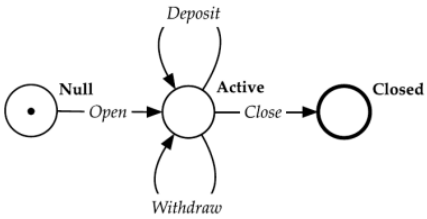

FIGURE 3.1

A state-transition diagram for a bank account atom. Once the account has been opened, deposits and withdrawals can be made in any order, until it is closed.

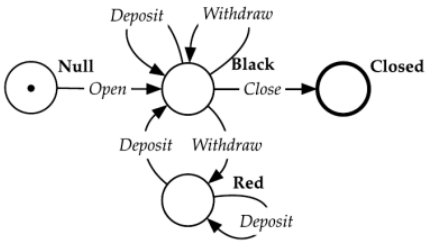

FIGURE 3.2

An alternative non-deterministic state-transition diagram for a bank account atom. If the account is in the *Red* state, withdrawals are not allowed.

It may help the reader to think of the dot as a **token** that is passed from state to state, e.g., when an *Open* event occurs, the token is passed from the *Null* state to the *Active* state; the *Null* state loses the dot, and the *Active* state gains it. The state containing the token is called the **current state**.

A path through a labelled graph is said to **spell** the sequence of the labels on its edges. One path from the initial state to the final state of Figure 3.1 spells the sequence [*Open*, *Deposit*, *Withdraw*, *Withdraw*, *Deposit*, *Withdraw*, *Close*]. There are many other paths from the initial to the final state, but they all begin with *Open* and end with *Close*. Between them can come any number of occurrences of *Deposit* and *Withdraw*, in any order. The possible sequences are called the **sentences** accepted by the automaton, and in general, the labels on the edges are called **letters**. For our purposes, letters and events are synonymous.

An FSA is an abstract notion, whose primary purpose is to define a set of sentences, called a **language**. A state-transition diagram may be used to describe an **acceptor**, an automaton that accepts or classifies input. If the input spells a sequence of transitions that takes the automaton from its initial state to one of its final states, it is said to **accept** the input, and the input is classified as a **valid sentence**. If the input does not spell a path from the initial state to a final state, it is **invalid**. (In a systems context, this usually indicates a human error.) An FSA may be used to classify sentences according to which final state it reaches — if any.

Alternatively, a transition diagram may describe a **generator**, an automaton that generates output. If the automaton moves from its initial state to a final state, then the path it follows generates a valid sentence. In the systems we are considering, the models will be acceptors; the real world will generate the sentences they accept.

In the case of an acceptor, if, in any given state, the input letter uniquely determines the following state, the diagram and its corresponding automaton are said to be **deterministic**. If more than one following state is possible, the diagram is said to be

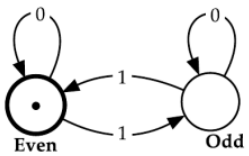

FIGURE 3.3

A state-transition diagram of an FSA to check the parity of a stream of bits. When it is in the *Even* state the parity of the stream received so far is even.

non-deterministic. In other words, it is deterministic if and only if the state-transition diagram defines a function from the input letter and the current state to the next state.

It is the systems analyst's task to decide what kinds of events and what states are relevant. In the case of the banking example, the analyst might, or might not, wish to distinguish between withdrawals that are refused because they would leave an account overdrawn, and those that don't. (See Figure 3.2.)

In the *Black* state, both deposits and withdrawals are possible, but if the amount of a withdrawal puts the account into the *Red* state, only deposits are allowed. In the case of Figure 3.1, a withdrawal might be dishonoured, but in *this* case, a withdrawal is not even a valid input. The amount of a deposit might or might not be sufficient to put the account back into the *Black* state.

3.2 DETERMINISTIC AND NON-DETERMINISTIC AUTOMATA

To illustrate some basic properties of finite-state automata, let us consider a simple problem: to determine whether the parity of a stream of binary digits is even (i.e., contains an even number of 1's). Figure 3.3 is the state-transition diagram of such an automaton. It has two states, labelled *Even* and *Odd*. It starts in the *Even* state. On an input of 1 it always changes state, but on an input of 0 its state doesn't change. If at the end of the input sequence it is in the *Even* state, the parity is even; if it is in the *Odd* state, the parity is odd. Since the goal here is to detect if the input has even parity, the *Even* state is its final state.[5]

As a more complex example, consider an FSA that accepts all sequences of binary digits whose *second* bit is a 1. State *U* is its initial state. (See Figure 3.4 on the next page.) The automaton moves to state *T* on the first bit of the input, irrespective of

[5] If, instead, both *Odd* and *Even* were final states, the automaton could be used to *classify* the input. The final state would indicate the parity.

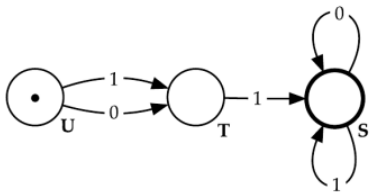

FIGURE 3.4

An FSA that accepts all sequences of bits whose second bit is a '1'. After the first bit, the FSA enters the state T. In state T, only a '1' bit is acceptable. If the bit is accepted, the FSA then remains in state S, its final state.

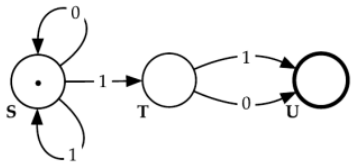

FIGURE 3.5

An FSA that accepts all sequences of bits whose second-last bit is a '1'. It is the reverse of the FSA of Figure 3.4. There are two edges leaving state S labelled 1. Therefore the FSA is non-deterministic.

whether the input is 1 or 0. On the second bit of the input, it moves to state S, but only if the input is a 1. If the input is a 0, the FSA rejects the sentence. S is its final state. Once in state S, the FSA remains in state S, irrespective of the sequence of bits that follows.

If we reverse all the edges of Figure 3.4 and exchange its initial and final states, we have an FSA that accepts the *reverse* of the original sentence, i.e., all binary sequences whose penultimate (second-to-last) bit is 1. (See Figure 3.5.) In this case, there is a difficulty: in state S, given an input of 1, it is impossible to know whether this will prove to be the second-to-last element of the input, so it is impossible to guess whether the automaton should move to state S or T.

This is an example of a non-deterministic finite-state automaton (**NFA**). Usually, an NFA must be converted into a deterministic finite-state automaton (**DFA**) before it can be useful. Even so, it remains possible to use the NFA itself, by keeping track of the *sets* of states that it can be in. In this case, in state S, on an input of 1, the NFA will move into a state that is a member of $\{S,T\}$. Only time will tell which is correct, so we must keep track of the set of possible states.

This idea also provides the key to converting an NFA into a DFA. We do this by mapping each distinct set of possible states of the NFA onto a *single state* of the DFA. The sets of states we consider are those that the NFA can reach after accepting the start of a valid sentence. We now demonstrate this idea by converting the NFA of Figure 3.5 into a DFA.

Before any input has been accepted, the NFA can only be in state S, so $\{S\}$ is its initial set of states, which will map to the initial state of the DFA.

Now consider the situation following the first digit of the input. If it is a 0, the NFA could only move from state S back to state S. Ambiguously, on an input of 1, the NFA can either move back to S or on to state T. Therefore, we have two transitions that are possible between *sets* of states,

$$\{S\} \underset{0}{\rightarrow} \{S\}$$
$$\{S\} \underset{1}{\rightarrow} \{S,T\}$$

We have now found a new set of states to think about, $\{S,T\}$. On input 0, the NFA can move from state S to state S or from state T to state U. On input 1, it can move from state S to either S or T, or it can move from state T to state U.

$$\{S,T\} \underset{0}{\rightarrow} \{S,U\}$$
$$\{S,T\} \underset{1}{\rightarrow} \{S,T,U\}$$

This time, we have discovered *two* further sets of states, $\{S,U\}$ and $\{S,T,U\}$.

In the set $\{S,U\}$, on input 0, the NFA can move only from state S to state S. On input 1, it can move from state S to either S or T. (In state U, it cannot move at all.)

$$\{S,U\} \underset{0}{\rightarrow} \{S\}$$
$$\{S,U\} \underset{1}{\rightarrow} \{S,T\}$$

In the set $\{S,T,U\}$, on input 0, the NFA can move from state S to state S or from state T to state U. On input 1, it can move from state S to either S or T, or it can move from state T to state U.

$$\{S,T,U\} \underset{0}{\rightarrow} \{S,U\}$$
$$\{S,T,U\} \underset{1}{\rightarrow} \{S,T,U\}$$

Fortunately, no new sets of states have appeared, so the above states are the only ones that can be reached.

We now construct a DFA by mapping each *set* of states of the NFA onto a corresponding state of the DFA. We can draw the state-transition diagram of the resulting automaton, which has four states in all. (See Figure 3.6.) Any set of states that includes U, the final state of the NFA, becomes a final state of the DFA.

Because we construct only one transition from each set of states for each input letter, we can be confident that the resulting automaton will always be deterministic, but can we be confident that the construction process will terminate? The NFA of Figure 3.5 has three states. There are only $2^3 - 1$ non-empty subsets of these states, so we can see that its corresponding DFA can have at most seven states. Therefore, the construction process must end after at most seven iterations.

This example clearly illustrates that, in pursuing an efficient implementation, synthesis (in this case from an NFA to a DFA) can lead to a result that is only obscurely related to its origin. The purpose of the NFA of Figure 3.5 is intuitive; the purpose of the DFA of Figure 3.6 is not.

In this example, it is easy to see why the resulting DFA needs at least four states: an obvious way to detect if the penultimate bit of a sequence is a 1 would be to store

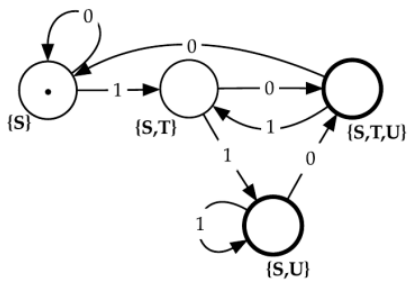

FIGURE 3.6

A deterministic state-transition diagram of an automaton that detects if the last but one bit of a binary sequence is a 1. Its states are labelled with sets of the states of the NFA of Figure 3.5.

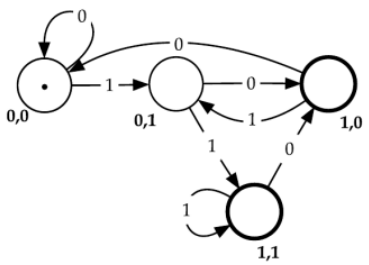

FIGURE 3.7

The unique minimal deterministic state-transition diagram of an automaton that detects if the second-to-last letter of a binary sequence is a 1. Its states are labelled with the most recent two bits of the input.

the most recent 2 bits of the input in a shift register (initially 00), and check its leading bit once the input is complete. A 2-bit shift register has precisely four states. Figure 3.7 shows the result of relabelling the states of Figure 3.6 as follows:

$$
\begin{aligned}
00 &= \{S\} \\
01 &= \{S,T\} \\
10 &= \{S,U\} \\
11 &= \{S,T,U\}
\end{aligned}
$$

In systems analysis, we rarely meet situations where an automaton is non-deterministic, but if we do, Algorithm 3.1 always lets us convert the resulting NFA to a DFA.[6]

In some cases, it may be more convenient to draw a state-transition diagram that appears to be non-deterministic, when actually it is not. Consider Figure 3.2, which shows the more detailed view of a bank account. The state *Red* indicates that the account is overdrawn, and the state *Black* indicates that the account is solvent. The event *Withdraw* can move the account from the *Black* state to either the *Black* or the *Red* state. In this example, it happens deterministically: if the amount of the withdrawal exceeds the current balance, the account becomes *Red*, otherwise it remains *Black*. The analyst has distinguished the *Black* and *Red* states to show that *Withdraw* is forbidden in the *Red* state. The deterministic nature of the FSA could have been shown by considering withdrawals that do or do not overdraw the account as two different kinds of event. Alternatively, the analyst could have combined the

[6] If the NFA has N states, the resulting DFA may have as many as 2^{N-1} states, so the problem is intractable in theory, but this is most unlikely to be an issue in practice.

Algorithm 3.1 Converting an NFA into an equivalent DFA.

1. Begin with an NFA N and a set of sets of states D initially containing a singleton set containing the initial state of N.
2. For each newly added set of states S in D, and each letter L in N,
   - Find the maximal set of states T such that an edge $V \xrightarrow[L]{} U$ is present in N, where $V \in S$ and $U \in T$.
   - Add the edge $S \xrightarrow[L]{} T$ to D.
3. Repeat Step 2 until no new sets of states can be added to D.
4. The sets of states in D are the states, and the edges in D are the transitions of the required DFA.

Red and *Black* states into one (as in Figure 3.1) and made a note that there are further conditions on whether a *Withdraw* event will be honoured. We cannot expect an FSA to express all the rules of a business.

3.3 REGULAR EXPRESSIONS

We can model an atom in two distinct ways: we can describe its behaviour, or we can describe how it changes its state. In a **behavioural description** we focus on what *sequences* of events are possible. In a **state-based description**, we take the view that the current state of the system is determined by prior events and that the current state of the system then determines what future events can occur. These descriptions are equivalent; we can derive behaviour from states or derive states from behaviour.

We can describe the possible sentences accepted by an automaton using a formula, called a regular expression, which uses three familiar constructs: *sequence*, *choice*, and *iteration*. We may have one thing followed by another, we may choose between alternatives, and we may repeat things any number of times. There is one limitation: regular expressions do *not* permit recursion. This isn't a problem, because real things are not recursive.[7] Although a component of a bicycle can be a part of a part of a part of a bicycle, etc., it can't be a part of *itself*.

We may define **regular expressions** formally as follows:

- If x is any letter (a transition), it is a regular expression.
- If R and S are regular expressions, $R\,;S$ denotes expression R *followed by* expression S.
- If R and S are regular expressions, $R \cup S$ denotes *either* expression R *or* expression S.

[7] Fiction is different: Dr Who once landed the TARDIS (which is bigger on the inside than on the outside) inside itself, creating an infinite regress.

- If R is a regular expression, R^* denotes a sequence of zero or more occurrences of R.
- ε (epsilon) denotes an empty expression, a path of length zero

where '$*$', binds tightest, '**;**' next, and '$\cup$' binds loosest.[8]

The language accepted by the bank account automaton of Figure 3.1 (page 82) is given by the regular expression

$$open\,;(deposit \cup withdraw)^*\,;close \tag{3.3.1}$$

A regular expression is a behavioural description: it says what sequences of letters can occur, without the need to invoke the idea of state. We might therefore prefer to use behavioural descriptions, because different automata can accept the same language. For example, the state-transition diagram of Figure 3.2 (page 82) defines an FSA that accepts exactly the same language as that of Figure 3.1 (page 82).

On the other hand, the expression

$$open\,;(deposit^*\,;withdraw^*)^*\,;close \tag{3.3.2}$$

describes the same language as expression 3.3.1, so neither a regular expression nor an FSA can claim to be a canonical way to describe a language.[9]

What we can say, in general, is that a given language can often be described by many possible regular expressions or by many possible transition diagrams. The good news is that nothing can be described using regular expressions that cannot be described using state-transition diagrams, and *vice versa*. Indeed, algorithms exist that enable each to be synthesised from the other.

The attentive reader will have noticed that the operators we have used here to construct regular expressions are the same relational operators we defined in Section 2.4.4. This is because each input letter defines a relation between states. Consequently, any theorem or transformation that is valid for binary relations is also valid for regular expressions.[10]

In the case of the shift-register DFA of Figure 3.7 (page 86), **0** and **1** are the following functions:

$$\mathbf{0} : States \to States = \{00 \mapsto 00, 01 \mapsto 10, 10 \mapsto 00, 11 \mapsto 10\}$$
$$\mathbf{1} : States \to States = \{00 \mapsto 01, 01 \mapsto 11, 10 \mapsto 01, 11 \mapsto 11\}$$

In theory, no other operators are required; any language can be described solely by the operators listed above. Even so, a language can often be described more concisely using additional operators:

---

[8] Yes, these are the same regular expressions used by utility programs such as *grep*. The difference is that *grep* uses a more convenient syntax. Nonetheless, any *grep* expression can be reduced to the form used here.

[9] But see the discussion of *minimal DFAs* in Appendix A.

[10] With this interpretation ε is the identity function.

- R^+ means the same as $R\,;R^*$ and denotes at least one occurrence of R.
- $R^?$ means the same as $R \cup \varepsilon$ and denotes an optional occurrence of R.
- $R \cap S$ is an expression that describes all sentences accepted by R that are also accepted by S.
- $R \setminus S$ is an expression that describes all sentences accepted by R that are *not* also accepted by S.

States and behaviours are closely connected. A state is both a summary of previous behaviour, and a restriction on future behaviour. If a bank account automaton is in the *Active* state (in Figure 3.1), it has accepted an *Open* event and zero or more *Deposit* or *Withdraw* events, but has never accepted a *Close* event. Likewise, it is ready to accept one or more *Deposit* or *Withdraw* events, or one *Close* event, but not an *Open* event. It has accepted the **phrase** *Open* ; $(Deposit \cup Withdraw)^*$ and is ready to accept the phrase $(Deposit \cup Withdraw)^*$; *Close*. In the *Null* state, it has accepted the empty phrase ε and is ready to accept a complete sentence; in the *Closed* state, it has accepted a complete sentence and is ready to accept ε.

The argument in favour of describing behaviour is that it can be done without the need for states. The argument *against* describing behaviour is that before we can actually implement a model, we usually have to derive states from behaviour. Here, we consider it more convenient to deal with states directly, and let the behaviour emerge. The correspondence between the behavioural and state-based approaches is therefore peripheral to our purposes, so it has been relegated to Appendix A, where we present well-known algorithms for deriving a state-based description from a behavioural description and *vice versa*. Even so, the reader is strongly advised to read Appendix A, because it illustrates several important points about synthesis methods in general.

3.4 FINITE MEANS FINITE

Before looking at some practical uses of FSAs, let us consider a simple problem that they cannot deal with: checking to see if the left and right parentheses in a text are correctly nested, i.e., that the numbers of left and right parentheses are equal, and that the number of right parentheses never exceeds the number of left parentheses. For example, '$((a,b),(3+4))$' is a valid sentence, but '$(2*3)+)9($' and '$(f(x)$' are not.

Figure 3.8 shows part of the state-transition diagram of such an automaton. When '(' is scanned, the machine moves one state to the right; when ')' is scanned, it moves one state to the left. When any other character is scanned, it remains in the same state. The states are labelled, logically enough, with the integers 0, 1, 2, 3, and so on. If at the end of the input, the numbers of left and right parentheses are equal, the machine should be in State 0. If the number of right parentheses ever exceeds the number of left parentheses, the machine has no possible transition, so the error is detected immediately.

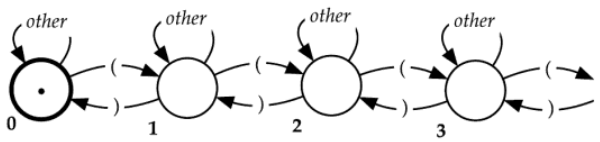

FIGURE 3.8

Part of the state-transition diagram of an automaton that detects if parentheses are correctly nested.

Unfortunately, it should be clear that if the automaton has a *finite* number of states, however large, an input sequence can start with so many left parentheses that the automaton runs out of states.

The reader will realise that such an automaton can easily be implemented by a counter, incremented for each '(', and decremented for each ')'. Nonetheless, if the representation of the counter is finite, eventually the counter will overflow, so the situation is like the state machine. Indeed, a counter *is* a state machine. We sum up this situation with the statement, 'FSA's can't count to infinity'.[11]

In practice, the FSAs we shall discuss will need only a handful of states. If we find a need for many states, we should think again. The sole purpose of our FSAs will be to keep track of what sequences of events make sense.

3.5 ANALYSING STATES

Consider the states of a copy of a *Book* from the point of view of a librarian. The important states of a copy might be: *On Order*, *Shelved*, *On Loan*, *Archived*, and *Scrapped*. From this point of view, although much detail has been omitted, the states of the book are discrete and finite in number. In contrast, a book retailer might consider a different set of states to be important: *On Order*, *In Stock*, *Sold*, and *Damaged*. Let us focus first on the librarian's view.

Libraries keep track of individual copies of books. The first action of the librarian would be to *Order* a copy from a supplier. After this, the copy would be *On Order*. Prior to this, the copy would be in its *Null* state, one that is of no particular interest to the library, but one that the state-transition diagram must include in order to show the *Order* event. Once a copy is *On Order*, the next event should be to *Receive* the copy, after which it is in the *Uncatalogued* state. The next action should be to *Catalogue*

[11] The reader will recognise that this task is similar to one performed by a compiler for a structured programming language in which statements and expressions can be nested to arbitrary depth. To overcome this limitation of FSAs, compilers use a *stack* of states, which can be infinitely deep in principle (although not in reality).

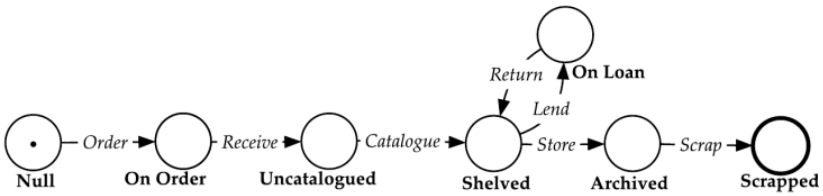

FIGURE 3.9

The state-transition diagram of a book, as seen by a librarian. Until a book is ordered, it is of no interest. Once it has been received, it is catalogued and made available for loan. Eventually, it may be archived or even scrapped. While the book is on loan, the only possibility is for it to be returned to the library.

the copy, after which it is said to be *Shelved*.[12] Once a copy is *Shelved*, a *Lend* event may occur, causing the state of the copy to become *On Loan*. In this state, a *Return* event will put the copy back into the *Shelved* state again. The resulting cycle means that a copy may be lent and returned any number of times. It is also possible for a *Store* event to change the state of a copy from *Shelved* to *Archived*. Eventually, a *Scrap* event may put the copy into the *Scrapped* state, which is a final state. The state-transition diagram of a library book is shown in Figure 3.9.

We shall have more to say about how states are represented later, but for the present, assume that the state of a copy is indicated by the value of a variable, a small integer, for example. Actually, *On Loan* is really a family of states, because a librarian cares who has borrowed each copy. Therefore, the representation of the *On Loan* state would need to be supplemented with the borrower's identity.

We may make a distinction between FSAs implemented by hardware and those implemented by software. Hardware FSAs have a permanent physical existence. The FSA to check the parity of a stream of bits can be implemented by a simple electrical circuit. On the other hand, software FSAs usually come into existence by being **instantiated**, i.e., memory is allocated to them as needed. Typically, when a library places a book on order, a record will be created to represent it. Prior to this, when the book is in the *Null* state, no record will exist.

It is wise to develop the transition diagram of an FSA further using a **state-transition matrix**. The DFA for a library book, as established so far, has the matrix shown in Table 3.1.

First, notice that if an FSA is deterministic, its current state (column) and the event (row) uniquely determine its next state. If this isn't the case, then the analyst should consider whether what appears to be one state or one kind of event is actually two or more.[13] For example, in an academic record system, a single event such as

[12] Libraries tend to claim that books are shelved even when they are sitting around *waiting* to be shelved. It is usually too much trouble to do anything else.

[13] As in the case of *Red* and *Black* bank withdrawals considered earlier.

Table 3.1 The state-transition matrix for a library book. The columns show existing states, the rows show events, and the cells show the new state following the event

	Null	*On Order*	*Uncatalogued*	*Shelved*	*On Loan*	*Archived*	*Scrapped*
Order	*On Order*	–	–	–	–	–	–
Receive	–	*Uncatalogued*	–	–	–	–	–
Catalogue	–	–	*Shelved*	–	–	–	–
Borrow	–	–	–	*On Loan*	–	–	–
Return	–	–	–	–	*Shelved*	–	–
Store	–	–	–	*Archived*	–	–	–
Scrap	–	–	–	–	–	*Scrapped*	–

Assess leading non-deterministically to the *Passed* state or *Failed* state could be subdivided into events, *Pass* and *Fail*, each leading deterministically to *Passed* and *Failed*. Alternatively, it may be more convenient to combine the *Passed* and *Failed* states into a single *Assessed* state. A third option is to allow the non-determinism to stand, with a footnote to the effect that the mark associated with the *Assess* event determines whether the new state is *Passed* or *Failed*. There is no right or wrong answer. The analyst will have to make a choice, probably based on requirements that will emerge later. For example, if the system must remember the marks that candidates obtain, it can then easily be established whether an *Assess* event is a *Pass* or *Fail*.

The most notable feature of the matrix of Table 3.1 is its predominance of empty cells. Most of these represent errors: for example, it makes no sense to borrow a copy that is already *On Loan*.[14] Even so, the analyst shouldn't rest until every cell has been accounted for. Can a copy be removed from the store and be re-shelved? Can a shelved copy be re-catalogued? What happens when a copy is lost by its borrower, stolen from the shelves, or damaged beyond repair? These issues can only be clarified by talking to a librarian, a domain expert. We call this process of discovery **event-state analysis**. Likewise, we may refer to a state-transition diagram as an **event-state diagram**, or a state-transition matrix as an **event-state matrix**. Event-state analysis is a two-way process: the model generates the questions, and the answers generate the model.

Apart from ensuring that all the cells are accounted for, it is important to ensure that no events or states have been missed. New events can often be discovered by asking questions such as, "When a copy is on the shelves, can anything happen to it apart from it being borrowed or archived?" New states can be discovered by questions like, "Are there different ways of borrowing a copy?" or, "Are there different kinds of

[14] This might result from the failure to record a *Return* event.

borrowers?" This might lead to the analyst's discovery of inter-library loans, resulting in a whole list of hitherto undiscovered events. In the case of physical objects, such as books, states often correspond to places. In this context, it is worth asking, "Apart from the shelves, the store, or the hands of a borrower, are there any other places where a copy can be?"

Finally, it is worth checking whether the final and initial states of atoms need to be distinct, since both states might correspond to a complete absence of data. For example, no record for a copy may exist at all until it is ordered. Perhaps, when it is scrapped, its record should be deleted, or maybe it should be retained for historical or statistical purposes. If its record *is* deleted, a scrapped copy looks no different from a copy that has never been ordered, and the record can be returned to the *Null* state. The record can be **destroyed**, i.e., the memory it occupies can be freed for reuse.

People make mistakes. A *Scrap* event might specify the wrong book, causing it to become *Scrapped*. No event we have considered can undo this action. To be able to reverse an erroneous event, a special, rarely used, **reversal event** may be introduced.

The analyst should also check that the state-transition diagram is strongly connected. This means that it has a path from every state to every other state, so *some* sequence of events can always restore an atom to any given state. For this purpose, the analyst may find an event-state diagram more useful than a matrix, as it is usually easy to see where a graph contains cycles. Alternatively, the analyst can reduce the event-state matrix to an adjacency matrix by ignoring the names of events and, from that, find its strongly connected components. (See Section 2.5.5.) One strategy (often used with hardware devices) is to provide a *Reset* event that restores an atom to its initial state. From there, a suitable sequence of events can drive it to any state that it can ever reach.

Care is needed: reversing an erroneous *Scrap* event by faking an *Order* and *Catalogue* event is likely to mean that the copy is paid for twice.[15] Reversing an event is typically more difficult, for example, than implementing *undo* in a word-processor. Erroneously recorded events may remain undiscovered for a long time, and it may be impractical to reverse every action that has occurred since the error was made. Sometimes too, an error can cascade, causing further errors that affect the snapshots of other atoms in the system. For example, erroneously recording that a book was loaned to one patron may not be detected until a second patron of the library actually borrows it. At this time, the system should baulk because the copy is already in the *On Loan* state. But by then, the first patron, unable to return the copy, may have been asked to pay a fine. Such errors can be hard to fix. As a result, it is tempting to allow an operator of a system to make arbitrary changes to states, but this is dangerous to do, especially if the changes they make go undocumented.

You are unlikely to hear about reversal events from a domain expert. Often, no formal system exists for dealing with mistakes, and each error is handled on an *ad hoc* basis. Nonetheless, persistent questioning can reveal some of the more common error-correcting strategies.

[15] Even so, this might be attractive because it is simple to implement. For example, the database could include an imaginary vendor who never charges for books.

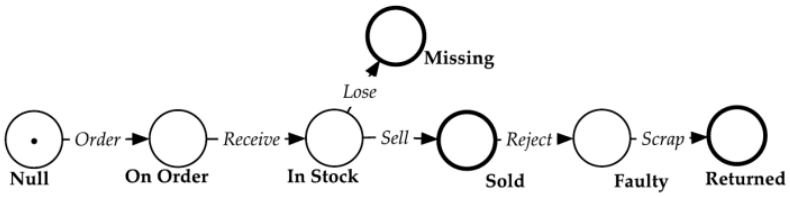

FIGURE 3.10

The event-state diagram of a book, as seen by a bookseller. There are three possible final states. A book may be sold, it may be missing (presumed stolen), or it may be returned to the publisher.

3.6 COUNTING

A bookseller's FSA for a book will be somewhat different from a librarian's. For a bookseller, a book's states might include *On Order*, *In Stock*, *Sold*, *Missing*, *Faulty*, and *Returned* (to the publisher). Its events might include *Order*, *Receive*, *Sell*, *Reject*, *Return*, and *Lose* (see Figure 3.10). However, the most important difference is that, unlike a librarian, a bookseller isn't interested in individual *copies* of books. A bookseller is interested in *titles*, and also whether the book is hardback or paperback, and so on. All books belonging to a given edition are alike. It makes no sense for a bookseller to track the progress of each copy of a book. Instead, it is the *number* of books in each state that matters.

This difference is reflected in the way states are represented. Instead of a variable that represents the state of an individual copy, a set of variables is needed, one integer to count the number of copies of the book in each state. When a copy moves from one state to another, for example, when a *Sell* event occurs, the number of copies *In Stock* decreases by 1, and the number *Sold* increases by 1. In other words, copies of books are *conserved*. If a copy is stolen, it is usually better practice to increment the number of books *Missing* than to merely decrement the number *In Stock*.[16] Similar conservation laws apply to virtually any system.

From this, we see that integers in a system often record the numbers of similar atoms in given states. By an extension of this idea, currency amounts represent the numbers of coins of some basic denomination: cents, pence, etc. Indeed, for many purposes, coins themselves have already become obsolete, and numbers have taken their place.

In an accounting system, coins (or their representations) can be in many states, one state for each account to which they might be credited. Each such account is

[16] Counting does not imply that we are no longer dealing with FSAs; the number of states of each atom (e.g., a book) is still quite small.

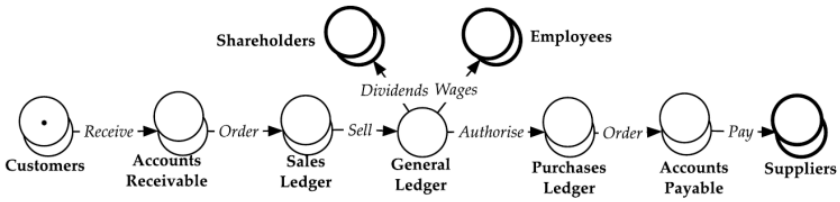

FIGURE 3.11

States of coins in a hypothetical accounting system. Only the main flows are shown.

typically represented by a separate record. Within the system, money is conserved. If the amount in one account increases, that in another decreases. That is the basis of double-entry bookkeeping. Indeed, because accounts can contain any amount of money, they are not FSAs and cannot be modelled by them. Consequently, although a coin's states are finite, a coin might be in so many states (one for each account) that drawing its event-state diagram might seem an impossible task. Fortunately, accounts fall into families that share similar behaviour, usually corresponding to ledgers. In drawing the event-state diagram for money, we should show families of states such as *Sales*, *Customers*, and so on. The *Accounts Receivable* ledger would be subdivided by customer, whereas the *Sales Ledger* might be subdivided by product. Figure 3.11 sketches part of an accounting system, in which money flows from customers to *Accounts Receivable*, and from there to a *Sales Ledger*, and so on. The states labelled *Customers* and *Suppliers* really lie outside the accounting system. Money is conserved *within* the system. When an *Order* event occurs, through the miracle of double-entry bookkeeping, money is transferred from *Accounts Receivable* to a *Sales Ledger*. On the other hand, when a *Receive* or *Pay* event occurs, marking a payment by a customer or a payment to a supplier, money crosses the system boundary and appears or disappears.

In an **open system**, conservation laws break down at its boundaries. Typically, no estimate is made of the number of atoms in a *Null* state. A bookseller doesn't care how many books have *not* been ordered. Numbers become important only when an order is placed.

A further extension of the conservation idea applies to **stuff**. Stuff cannot be counted; it has to be measured. A company that sells oil cannot be expected to count every molecule. Instead, it estimates the number of molecules it sells by measuring the volume sold.[17] Again, *stuff* is usually conserved, and the oil company will need to have a measure of the volume of fluid in each state, including, perhaps, the amount that has leaked or spilled.

Sometimes, a system's view of atoms changes; it may view the same atoms as sometimes interchangeable, and as sometimes distinct. A librarian might merely

[17] Because of thermal expansion, fluids really ought to be sold by mass rather than by volume.

count the number of books of a given edition while they are *On Order*, books only becoming distinct when they are *Catalogued* and given copy numbers. The books *On Order* may be represented by numbers, but those on the shelves may be represented by individual records. Similarly, a car manufacturer might regard all cars of a certain model as indistinguishable until they are stamped with their chassis and engine numbers.

Irrespective of whether the system keeps track of individual atoms, indistinguishable sets of atoms, money, or stuff, the story of every atom, coin, or molecule can be modelled by an event-state diagram. Despite this, ***and this is important***, the atoms (e.g., coins) that are modelled may not correspond one-to-one with objects or records in an information system; it may be their states (e.g., accounts) that are represented.[18] Trying to draw an event-state diagram for a *state* is an impossible, meaningless, task.

3.7 CONTINUOUS SYSTEMS

Although continuous systems aren't the focus of this book, it is worth giving some space to them. As we shall see in a later chapter, both discrete and continuous systems exhibit **dynamic behaviour**. Such behaviour can be **autonomous**, that is, it can occur without external input. Sometimes autonomous behaviour is desirable, and sometimes it is not.

We cannot apply the notion of an FSA to the continuous behaviour of an aircraft, so what alternative do we have? Such a subject would fill a book, so we turn instead to a simple pendulum.

Consider an ideal pendulum consisting of a point mass m hanging by a weightless string of length l. Let us assume that the mass is displaced a distance x from the vertical. Intuitively, we can see that a force acts on the mass to restore it to the vertical, and that the mass will therefore accelerate. Let us assume that it moves with velocity v. We assume further that the pendulum is *perfect*, i.e., that it doesn't lose energy through friction.

We suggested earlier that systems obey conservation laws. What is conserved here is energy. The point mass has two forms of energy, kinetic energy and potential energy, and their sum is constant. The pendulum's kinetic energy is given by the formula $\frac{1}{2}mv^2$. The pendulum's potential energy is given by mgh, where g is the acceleration due to gravity, and h is the vertical displacement of the mass above its lowest position. For small displacements $h = \frac{1}{2}x^2/l$. Therefore $\frac{1}{2}mv^2 + \frac{1}{2}mgx^2/l = E$, where E is the (constant) energy in the system.

The state of the pendulum is defined by its position and velocity. If we plot position, x, horizontally and velocity, v, vertically, then with suitable scaling, we see that all the points with energy E must lie on a circle whose area is proportional to E, as shown in Figure 3.12. This is actually a form of state-transition diagram. The

[18] This is why we don't call atoms 'objects'. This term is used in object-oriented programming to represent some data and its associated methods. Often, an atom can be represented by an object, but not always. What is more, an object often represents a set of atoms, rather than an individual atom.

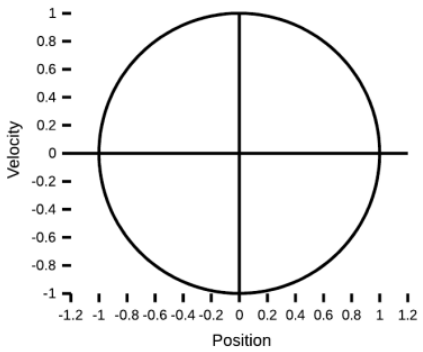

FIGURE 3.12

The state trajectory of a perfect undamped pendulum. The scaling is normalised so that the initial position of the pendulum is 1.0, and its maximum velocity is 1.0. Its state moves from the point (1.0, 0.0) clockwise around the circle, once for each complete swing.

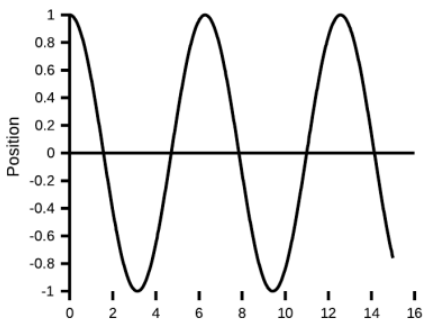

FIGURE 3.13

The position of a perfect pendulum as a function of time. Its initial position is normalised to +1.0. Each horizontal unit is normalised to represent $\sqrt{(l/g)}$. The pattern repeats at intervals of 2π.

state of the pendulum moves around the circle. The states lie infinitesimally close to one another, linked by infinitesimally small transitions. The circle is the pendulum's **state trajectory**. Figure 3.13 shows the oscillatory displacement of the pendulum as a function of time.[19]

[19] An analysis using differential calculus will tell us a little more: to counter the weight of the mass, the string must exert a vertical force equal to mg. For small displacements, this will have a horizontal component $f = -mgx/l$. The horizontal acceleration of the mass is therefore $a = f/m = -gx/l$. Mathematically, acceleration is the second differential of position, so that

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\frac{g}{l}x \tag{3.7.1}$$

In the case that the pendulum has zero velocity and initial displacement x_0, this differential equation has the well-known solution

$$x = x_0 \cos\left(\sqrt{\frac{g}{l}}t\right) \tag{3.7.2}$$

Differentiating with respect to t, we see that

$$v = \sqrt{\frac{g}{l}}x_0 \sin\left(\sqrt{\frac{g}{l}}t\right) \qquad \text{and} \qquad a = -\frac{g}{l}x_0 \cos\left(\sqrt{\frac{g}{l}}t\right) \tag{3.7.3}$$

So $a = -(g/l)x$, as required. Since the cosine function repeats after an interval of 2π, the oscillation has period $2\pi\sqrt{g/l}$.

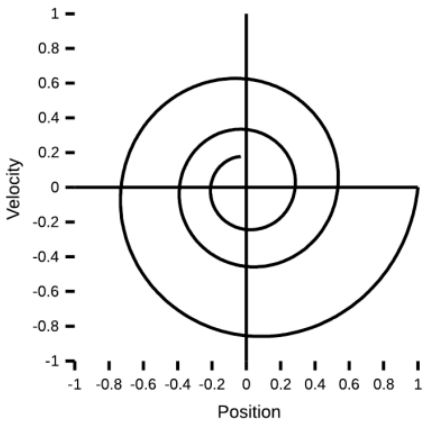

FIGURE 3.14

The state-transition diagram of a damped pendulum. Because energy is lost, the oscillations decay.

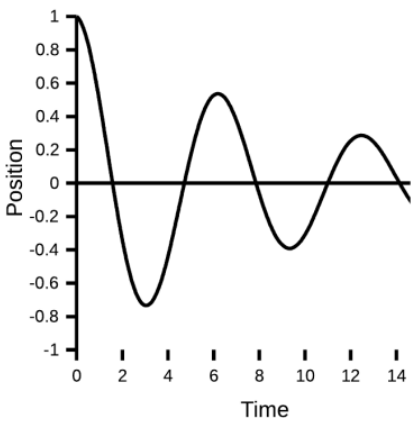

FIGURE 3.15

The damped oscillation of a pendulum. The amplitude of the oscillations decays with time.

In practice, since any real pendulum will lose energy to its environment, its state trajectory will be a spiral converging (asymptotically) towards the origin, as in Figure 3.14, and its oscillation will decay exponentially, as in Figure 3.15.

To extend these ideas to the analysis of an aircraft, we have to remember that its state is described by three linear positions and velocities, and three rotational positions and velocities. Therefore we would need to visualise its state trajectory in 12 dimensions. The equations that describe its motion are complex. Even so, an aircraft does have one mode of behaviour, the **phugoid oscillation**, which is roughly similar to the case of the simple pendulum.

Imagine an aircraft that is flying straight and level, but a little too slowly. Because of this, its wings won't generate enough lift to support its weight. The aircraft will start to descend, gaining speed as it does so. The additional speed will generate more lift, so the aircraft will climb again. At the top of the climb, it will again be flying too slowly, and start to descend again. Like the pendulum, the aircraft exchanges potential energy (height) for kinetic energy (speed).[20] Fortunately, this kind of oscillation is usually slow, easily controlled by the pilot, and even without intervention, like the swing of a pendulum, it gradually decays.[21]

[20] In theory, the height of an aircraft should be described by a real number of infinite precision. In practice, continuous systems are usually modelled digitally to finite precision. A 64-bit word has 2^{64} (almost 10^{20}) states, enough to represent the height of an aircraft far more precisely than it can be measured. Likewise, we must approximate the continuous flow of time by a series of sufficiently frequent snapshots. An airliner travels less than 0.3 millimetres per microsecond, so it isn't hard to compute sufficiently frequent approximations for practical purposes.

[21] In highly manoeuvrable fighter aircraft, the pilot's intervention can make things worse.

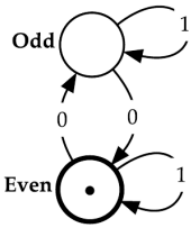

FIGURE 3.16

An FSA to check for an even number of 0s.

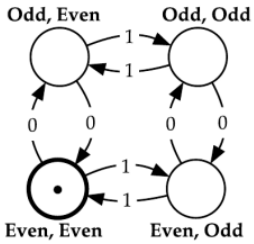

FIGURE 3.17

An FSA to check for an even number of 0s and an even number of 1s.

Similar dynamic behaviour is found in business systems and national economies: suppose you go to the supermarket only to find that your favourite blend of coffee is out of stock. Realising that its supply is unreliable, the next time you visit, you take twice your normal amount. Multiply this by a dozen like-minded individuals, and the supermarket manager quickly notices how fast the coffee is selling, and therefore orders more than usual. Shortly, the shelves are full of your favourite coffee, but you still have a spare supply at home and don't need more. At this point, the supermarket manager notices that the coffee isn't selling after all — and you can continue the story yourself.

3.8 PRODUCTS OF FSAS

As a preparation for understanding shared events, recall the FSA of Figure 3.3, which checks that a sequence of bits contains an even number of 1s. Figure 3.16 describes a similar FSA that checks whether such a sequence contains an even number of 0s. If both FSAs share the same events, together they will check that the input contains even numbers of 1s and 0s. We can describe their combination in a single FSA that has states $(Even, Even)$, $(Even, Odd)$, $(Odd, Even)$, and (Odd, Odd); the set of states is the Cartesian product of the states of the FSA of Figure 3.3 and the FSA of Figure 3.16. (See Figure 3.17.)

Clearly, the number of states of the FSA of Figure 3.17 is the product of the numbers of states of its component FSAs. If we collapse the diagram vertically, merging $(Even, Even)$ with $(Odd, Even)$ and $(Even, Odd)$ with (Odd, Odd), we recover the FSA of Figure 3.3, and if we collapse it horizontally, we recover the FSA of Figure 3.16.

As we discussed earlier, we may express the behaviour of an FSA using a regular expression. An expression that describes any sequence containing an even number of 1s is $(0 \cup (1 \,;\, 0^* \,;\, 1))^*$. Similarly, $(1 \cup (0 \,;\, 1^* \,;\, 0))^*$ describes any sequence containing an even number of 0s. The FSA of Figure 3.17 therefore accepts the expression

$(0 \cup (1\,;0^*\,;1))^* \cap (1 \cup (0\,;1^*\,;0))^*$, where the intersection operator means that a valid sentence must satisfy both $(0 \cup (1\,;0^*\,;1))^*$ and $(1 \cup (0\,;1^*\,;0))^*$.[22]

3.9 SHARED EVENTS

Real-world events often affect more than one atom. Borrowing a book from a library affects the state of the book and also the state of the borrower. We call such interactions **shared events**.

Often, a shared event involves a **rate of exchange**. Selling a book results in a loss of stock to the bookseller, but also an increase in money. The rates of exchange between books and money are prices. Rates of exchange are a second way in which numbers are important in a system.

The main point about shared events is that all parties to the event have to be in states where the event is permitted. That is so important, I shall say it again:

> ***Each atom involved in a shared event must have an event-state diagram that includes at least one edge labelled with that event, and the event itself is only valid if* all *the atoms concerned are in states that have such edges leaving them.***

Let's look at a simple student enrolment system. When a candidate enrols in a subject, two atoms participate in the event: the *Candidate* and the *Subject*.[23] For this to happen, the candidate must be *Admitted* and the subject must be *Offered*. Figure 3.18 shows the event-state diagram for a *Candidate*, and Figure 3.19 shows the diagram for a *Subject*. We see that once a candidate has been admitted to a degree, he or she may *Enrol* in subjects or *Withdraw* from them, until the candidate *Graduates*. Likewise, once a subject is *Offered*, candidates may *Enrol* in it or *Withdraw* from it, until some date at which enrolments become *Frozen*, after which candidates enrolled in it will be *Assessed*. As a result, a candidate cannot, for example, enrol in a subject that has been frozen, and a candidate who has already graduated cannot enrol in any subject.

Even so, these constraints aren't strong enough: the diagrams allow a candidate to *Withdraw* from a subject before ever enrolling in it. Neither the *Candidate* DFA nor the *Subject* DFA prevents this.

We now consider a corollary to the point we just made: the *only* things that determine whether an event can occur are the states of the atoms that share it. This is again so important I shall say it again:

> ***Unless the state of something prevents it, an event is valid.***

[22] Regular expression syntax doesn't usually include the ∩ operator, because it can always be eliminated. Unfortunately, eliminating it is an NP-complete problem, and the resulting expression can be ridiculously long.

[23] Actually, *three* atoms participate, as we shall soon see.

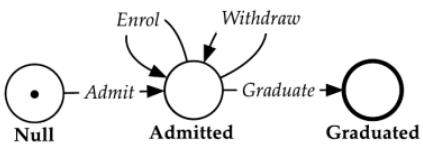

FIGURE 3.18

The event-state diagram for a *Candidate*.

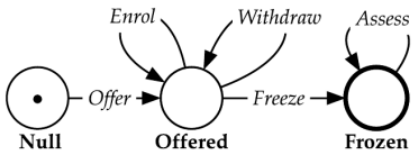

FIGURE 3.19

The event-state diagram for a *Subject*.

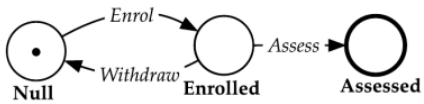

FIGURE 3.20

The event-state diagram of an *Enrolment*.

Can we adjust the *Candidate* and *Subject* DFAs to prevent silly sequences of events being possible? One approach would be to count the number of subjects in which a candidate is enrolled and to count the number of candidates enrolled in a subject. Clearly, this will prevent a candidate withdrawing before enrolling and will stop the number of candidates in a subject becoming negative. Despite this, other errors are still possible. Candidate C could enrol in subject S and candidate D could enrol in subject T, then C could withdraw from T, despite never having enrolled in it. Clearly, we must prevent this.

That we are discussing *counters* suggests that we are regarding some set of atoms as indistinguishable. However, the *number* of subjects in which a candidate is enrolled is secondary to *which* subjects they are. We would be wrong to regard subjects as indistinguishable. Similarly, to enable a candidate to withdraw from a subject, we need to know the *set* of candidates in the subject, not just the number of candidates.[24] We would be equally wrong to regard candidates as indistinguishable.[25] *What we need is a new kind of atom to constrain the order in which events can occur.*

The atom concerned is already known as an *Enrolment*. An *Enrolment* is a record of the relationship between a particular candidate and a particular subject. Figure 3.20 shows its event-state diagram.

[24] It would be possible, but unwise, to extend the set of *Candidate* states to include a state for every combination of classes the candidate enrolled in or to extend the set of *Subject* states to include a state for every possible class list. Even as few as 10 possible classes would require a candidate to have over 1,000 states, and 100 candidates would need a subject to have over 10^{30} states. Neither possibility really respects the notion of a DFA.

[25] Despite this, we shall later learn that it remains convenient to keep track of the number of candidates enrolled in a subject (*Filled*), because there is a limit on the number of places available (*Quota*).

When a candidate wishes to *Enrol* in a subject, this is now a three-way interaction: the candidate must be in the *Admitted* state, the subject must be in the *Offered* state, and their associated enrolment must be in the *Null* state. The effect of an *Enrol* event is that the enrolment moves to the *Enrolled* state. A *Withdraw* event moves the enrolment back to its *Null* state; its effect is as if the *Enrol* event had never happened. In the *Enrolled* state, an *Assess* event moves the *Enrolment* DFA to the *Assessed* state. Whether *Withdraw* or *Assess* events are allowed is controlled not by the *Enrolment* DFA itself, but by the state of the *Subject* DFA.

A convenient, but perhaps confusing, convention has been observed in labelling the DFAs. If an event is shared with other DFAs, it must be taken into account when considering *each* DFA. For example, the *Subject* DFA allows for *Enrol* and *Withdraw* events only in the *Offered* state and not in the *Frozen* state. The absence of transitions for *Enrol* and *Withdraw* in the *Frozen* state implies that they aren't permitted. On the other hand, the absence of any transition for *Admit* events doesn't mean they are forbidden, because this isn't an event that *Enrolment* shares with any other DFA.

We must remember that each DFA is really one of a family of similar DFAs. There is a *Candidate* DFA for every candidate, a *Subject* DFA for every subject, and several *Enrolment* DFAs for every candidate or every subject.

Admittedly, a real student enrolment system is much more complex than this discussion suggests. For example, a candidate may be forbidden to enrol in certain subjects until having passed examinations in certain others. Likewise, complex rules may determine when a candidate may graduate. Although such rules are not theoretically beyond the scope of DFAs, there are better ways to deal with them. Such rules are typically embedded in computer procedures, in (human) experts, or in expert systems.

If we were to try to draw the state diagram of the product of the *Candidate* FSA of Figure 3.18, the *Subject* FSA of Figure 3.19, and the *Enrolment* FSA of Figure 3.20, we would obtain a three-dimensional FSA that might have as many as $3 \times 3 \times 3 = 27$ states. Mathematically speaking, drawing such a diagram is intractable, the number of its states being exponential in the number of its component FSAs. Fortunately, interactions between more than three FSAs are rare, and because of shared events, the number of reachable states is usually only a subset of the Cartesian product. In this example, only fifteen states can be reached from the initial state (see Table 3.2). Each state of the product consists of a triple: (*Candidate*, *Subject*, *Enrolment*). The first three columns show the value of the triple before an event, the fourth column shows a possible event, and the final three columns show the value of the triple (the new state) after the event.

Table 3.2 was derived as follows: the first three columns of the first two rows show the initial state. Only *Admit* and *Offer* events are possible. The *Enrolment* DFA cannot move from its initial state because *Enrol* events are blocked. The effect of an *Admit* event is considered in the first row, and the effect of an *Offer* event is considered in the second. The new state resulting from the second row is considered in the third and fourth rows. The new state resulting from the first row is considered in the sixth and seventh rows. Each new state generated in the last three columns is transferred to