The Automatic Design of Batch Processing Systems

by

Barry Dwyer, M.A., D.A.E., Grad.Dip.

A thesis submitted for the degree of
Doctor of Philosophy
in the Department of Computer Science
University of Adelaide

February 1999

Abstract

Batch processing is a means of improving the efficiency of transaction processing systems. Despite the maturity of this field, there is no rigorous theory that can assist in the design of batch systems. This thesis proposes such a theory, and shows that it is practical to use it to automate system design. This has important consequences; the main impediment to the wider use of batch systems is the high cost of their development and maintenance. The theory is developed twice: informally, in a way that can be used by a systems analyst, and formally, as a result of which a computer program has been developed to prove the feasibility of automated design.

Two important concepts are identified, which can aid in the decomposition of any system: 'separability', and 'independence'. Separability is the property that allows processes to be joined together by pipelines or similar topologies. Independence is the property that allows elements of a large set to be accessed and updated independently of one another. Traditional batch processing technology exploits independence when it uses sequential access in preference to random access. It is shown how the same property allows parallel access, resulting in speed gains limited only by the number of processors. This is a useful development that should assist in the design of very high throughput transaction processing systems.

Systems are specified procedurally by describing an ideal system, which generates output and updates its internal state immediately following each input event. The derived systems have the same external behaviour as the ideal system except that their outputs and internal states lag those of the ideal system arbitrarily. Indeed, their state variables may have different delays, and the systems as whole may never be in consistent state.

A 'state dependency graph' is derived from a static analysis of a specification. The reduced graph of its strongly-connected components defines a canonical process network from which all possible implementations of the system can be derived by composition. From these it is possible to choose the one that minimises any imposed cost function. Although, in general, choosing the optimum design proves to be an NP-complete problem, it is shown that heuristics can find it quickly in practical cases.

Declaration

This thesis contains no material that has been accepted for the award of any other degree or diploma in any university or other tertiary institution. To the best of my knowledge and belief, it contains no material previously published or written by any other person, except where due reference is made in the text.

I give consent for this copy of my thesis, when deposited in the University Library, to be available for loan and photocopying.

Barry Dwyer
February 1999

Acknowledgments

Although I have to admit that I have learned a lot in the process, completing this thesis has taken me an excruciatingly long time. I am therefore grateful for the faith and patience of my supervisor, Prof. C.J. Barter. I am even more grateful for the support of my wife, Linda, who has often had to forego the pleasure of my company while I worked on it. In addition, I would like to thank those of my friends with whom I have discussed my work. Even if they thought they weren't helping me, the act of explaining my ideas helped me clarify many difficult issues. Finally, I would like to thank the reviewers of my thesis for suggesting how it could be improved. I have done my best to follow their suggestions faithfully.

List of Figures

1. Introduction

1.1 The Aim of the Thesis

Consider the process graph of Figure 1.1.1, in which the rectangles represent processes, and the edges represent data-flows connecting them. Data flows in the direction of the arrows. Data flows between A and B in both directions; they are said to be 'strongly connected'. Data flows between B and C in one direction only; they are said to be 'separable'. Data doesn't flow between C and D at all; they are said to be 'independent'. A and D are separable, not independent. There is a directed path from A to D via B.

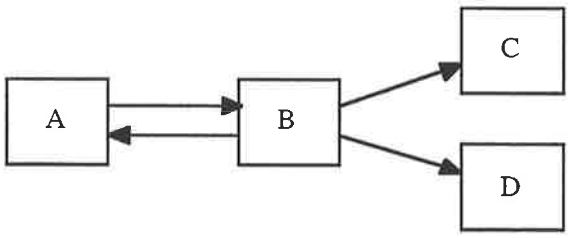

FIGURE 1.1.1: A PROCESS GRAPH

Under certain general assumptions, these simple distinctions turn out to be all-important in the design of systems, for the following reasons: Process B can be linked to process C by a queue, allowing flexible scheduling of the two processes. One option is for B and C to execute concurrently, with the queue implemented as a buffer, smoothing fluctuations in demand. Another is to allow C to execute on a different day of the week from B, with the queue implemented as a sequential file. Processes C and D can be scheduled even more flexibly, with C before D, C after D, or both executing in parallel. However, A and B can only execute in close synchronisation, no queues can be used, and little useful parallelism can be achieved.

These distinctions are useful in the design of systems of many kinds. They certainly prove important for the design of information systems. Chapter 3 shows how the parallelism obtained by exploiting separability and independence allows massive speed-up. High throughput is needed in major banks and clearing houses, which may process of the order of 10 to 100 million transactions overnight. This demands a throughput of up to 3,000 transactions per second. Given that current technology allows less than 100 updates to a hard disk per second, one must clearly exploit parallelism — or some equivalent technique — for such systems to succeed. It turns out that an equivalent technique indeed exists, and has existed for a very long time. The technique is known as batch processing.

Because of their efficiency, batch systems are the heavy production lines of the information processing industry. They are costly to design and implement, yet there is no existing formal theory underlying their construction. The aim of this thesis is to provide that foundation, and to show that it is feasible to develop tools to generate complete batch systems from specifications. Such tools would revolutionise a major programming activity by making batch processing systems cheap to write. This in turn would make the high efficiency of batch systems affordable to a wider class of users.

It has been estimated that over 150 billion lines of Cobol code are in current use today, and about 5 billion lines are added each year [Saade & Wallace 1995]. Cobol accounts for over 50% of the application code being written. Since interactive database applications are handled so much more easily by database management systems, it is fair to assume that most Cobol code implements batch processing systems.

This thesis makes two main contributions to the study of batch processing: it describes a formal technique that can be used to design batch systems from specifications, and it shows how its basic ideas can be adapted to implement massively parallel contention-free database systems. Here, a 'design' means a network of processes, similar to Figure 1.1.1. Collectively, the processes implement the requirements of the specification. The thesis goes on to describe a program that can design systems automatically, automating a task that presently requires the skills of an experienced systems analyst. (The remaining claims of the thesis are listed in Section 11.4.)

System design is currently a black art. To anyone with experience of designing batch information systems, an ability to design them automatically may seem a remarkable claim. This is not to deny that there are useful design methodologies that can assist a designer, but it is also accepted that they need the designer to have a good measure of intuition or experience. The better ones, [DeMarco 1978, Jackson 1983] for example, present the design process as a series of transformations, from problem to solution. The designer has merely to choose the correct series. Unfortunately, there are many design choices, so the design problem is combinatorially complex. Design must proceed with a sense of direction and purpose, which currently only an experienced human designer can provide.

Existing methodologies consider design basically as a process of decomposing the specification into component processes. No tractable algorithm is known that can design a batch system by decomposition. In contrast, the thesis treats design as a process of composition from 'primitive processes'. Every specification has a canonical decomposition into primitive processes. All feasible designs may be derived from this canonical form by pair-wise compositions. One way to express the difference is to say that the new approach is 'bottom-up' rather than 'top-down'.

It is admitted that choosing the best of these designs is also, mathematically speaking, an intractable problem, but the thesis demonstrates that simple heuristics can usually determine the best solution in a reasonable time. Some specifications can lead to highly efficient designs, some can only lead to inefficient ones. The thesis shows that this is an inevitable consequence of the specification, not a question of the designer's ingenuity.

Given a tool that can derive system designs from specifications, what becomes of the role of the systems analyst or designer?

It turns out that the canonical decomposition of a batch system depends on small details of its specification. These details may have a major impact on the performance of the resulting system. It becomes the designer's role to understand how these details affect efficiency and to choose the specification accordingly. In one case the designer might adjust the details of an algorithm, in another the designer might change the way facts are represented in the database, in a third case the designer, taking the role of systems analyst, might have to negotiate with the client to change a system requirement.

In fact, there is nothing new here; system designers already do these things. However, at the present time, they have no theory to back them up. There are two practical consequences of this lack: One is that errors creep into system designs, which only become apparent during programming or debugging, and which are sometimes impossible to correct without completely redesigning the system. The second is that although individual transactions are processed correctly, subtle processing errors arise involving interacting *sets* of transactions, so that, for example, a query may observe the database in an inconsistent state. Sometimes these errors are latent for long periods, only being discovered after many months or years of operation. In contrast, the methods of this thesis are based on a strict notion of correctness, referred to as 'real-time equivalence', which guarantees that the design is error-free — or at least, no more erroneous than its specification.

1.2 Transaction Processing Systems

The two key elements of a transaction processing system are a database, which stores information, and transactions, which update the state of the database. A third element is that the database usually models some real-world situation. Questions about the real world can then be answered by inspecting the model. It is typically cheaper to inspect the model than it would be to observe the real-world system itself. For example, a bank can check its financial situation simply by consulting its books — it does not have to count all the money in its vaults. Indeed, if it were not for modelling cash by entries in ledgers, a bank would have to keep every customer's money in separate boxes. In this case, the model actually replaces an aspect of the real world. There are some situations where only a model will do; for example, an airline reservation system is concerned with predicting which passengers will occupy aircraft seats in the future — the future cannot be observed any other way than by modelling it.

Transaction processing systems have long been responsible for driving the development of information processing technology. Indeed, it appears that the Sumerians invented writing precisely to keep track of business transactions [Friberg 1984]. Transaction processing systems were quick to exploit the invention of the Hollerith punched card machine in the 1890's, and drove its subsequent development — which became integrated with that of early computers — until the late 1970's. More recently, through bank teller machines, EFTPOS, airline reservation systems, and so on, transaction processing systems have become geographically dispersed, and have fuelled a demand for faster data communications.

Transaction processing systems should guarantee the four so-called ACID properties [Haerder & Reuter 1983, Krishnamurthy & Murthy 1991, Gray & Reuter 1992]:

Atomicity: Failed transactions do not produce side-effects: either a transaction completes its action on the database, or it has no effect.

Consistency: A transaction moves the database from one consistent state to another.

Isolation: A transaction depends on other transactions only through the changes they make to the database.

Durability: After a transaction completes, the changes it made to the system state persist indefinitely.

Even if a system lets many transactions be processed concurrently, these properties ensure that its current state may always be derived from its initial state by applying transactions one by one. This property is called 'serialisability'. Serialisability greatly simplifies the semantics of transaction processing by guaranteeing that the effect of a transaction depends only on the state of the database, and not on any concurrent transactions. Of course, transactions may still interact. If one person books the last seat on an aeroplane, a second person cannot also book it. However, the interaction occurs purely because the first transaction makes the number of free seats in the database become zero, so, to a first approximation, neither transaction needs to be aware of the other. But it is only to a first approximation, because, if the two transactions are concurrent, they must actively avoid interaction in some way. Typically, the first transaction will lock part of the database, the second transaction will detect the lock, and back off.

1.3 Batch Systems

There are broadly two kinds of transaction processing system, on-line and batch. In an on-line system, transactions update the database as the real-world changes that they model actually occur. In some cases they may record a change soon after it has taken place; in others they may even be a contractual part of the change — you can't book a seat on a plane or at the theatre unless a computer model shows that it is free. A batch system is one where the database is updated less often; the transactions are recorded as they occur, but the database is not changed until later. But when it is changed, a whole batch of recorded transactions is processed at one time. In batch systems, it is typically unnecessary to inspect the database to satisfy the contractual requirements. Either the update can proceed without regard to the state, or more usually, there is physical evidence that the transaction is valid. For example, if you want to buy a loaf of bread at the supermarket, it is enough to present the loaf at the checkout; the checkout operator does not have to consult an up-to-date inventory database to authorise the sale. The eventual motive for updating a batch system is usually that someone wants to inspect the current state of the system. For example, a supermarket may update its inventory database from its sales transactions when it wants to determine what new supplies it needs.

Why should we ever prefer to be lazy? Why not use on-line updating in every situation?

If one wants to make just one thing, it is best to bring the tools to the work. But if one wants to make a great many motor cars, radios or wrist-watches, it pays to bring the work to the tools. Activity centres around a line of workstations that perform single tasks, and the construction of any given product becomes fragmented, with jobs in progress sometimes waiting for long periods between stations. It is characteristic of production lines that work is not always carried out in the most 'logical' order, but in an order dictated by optimising the efficiency of the line. Indeed, it is often the case that the design of the product is influenced by the means of its production, so that it can be produced in the fewest steps.

In information systems, a few transactions are most efficiently processed by bringing the data to the transactions, but if there are many transactions, they are best processed by bringing the transactions to the data. As a simple illustration, suppose one needs to check a list of words against a dictionary. If the list is short, it is adequate to find each word at random; but if the list is long enough, it pays to sort it into alphabetical order, then scan the dictionary systematically from A to Z. This basic idea was exploited before the computer era, and early computer systems relied on it heavily. Sequential access media, such as card files and tapes, were once all that were available, so files had to be scanned in sequential order. Since the 1960's, random access storage devices, typically magnetic disks, have continued to decrease in price and improve in performance, and sequential access is no longer a necessity. Indeed, processing transactions one at a time using random access is usually the method of choice, because it gives immediate feedback and 'instant' results. Despite this, because of its superior efficiency, processing large batches of transactions using sequential access still has a place.

A system may be said to be interactive if its user does not switch to other activities while it is in use. To be considered interactive, a system should have a response time of less than 5 seconds. Traditionally, batch processing has been non-interactive; efficient batches typically contain a day's or a week's transactions. Generally speaking, the more transactions there are in a batch, the greater the efficiency. However, as Chapter 3 will show, when the throughput of a system is very high, efficient batches can be processed at least once per second.

1.4 The Macrotopian Reference Library

Consider the evolution of the (fictitious) Macrotopian Reference Library's loans system as it grows to meet increasing demands. The library does not lend to patrons directly, but only to its branch libraries. Loans are requested by telephone, by post, or by electronic mail. The loans system records the numbers of copies of books remaining on the shelves, and the number borrowed by each user library. Because there is an inevitable delay in delivering the books themselves, the system does not need to respond interactively.

Each book and each user is associated with an index card. When a user library borrows a book, the book record must be adjusted to decrement the number of copies on the shelves, and the user record must be adjusted to increment the number of books borrowed. The library has a rule that the last copy of a book must always remain on the shelves, so that it can be consulted by visitors to the library. However, such visitors cannot themselves borrow books.

In the simplest possible implementation of the system, one library clerk receives the orders, and adjusts both sets of records as each loan is made. The clerk must first check that the book being borrowed is not the last, then increment the number of books borrowed by the user and decrement the number of copies of the book remaining on the shelves.

Suppose that, as the demand for the library's services increases, the clerk cannot cope with the work load. During busy periods, the clerk may then merely record the identifiers of the books and users, and later, during quiet periods, update the file cards from the batches of loan records. Of course, with such an arrangement, the clerk must be careful not to issue a book before checking that one will still remain on the shelves. The clerk will soon find that it is more efficient to sort the loans into card file order, first by title to update the book card index, then by the user identifier to update the branch library cards, rather than to search the book and branch library files at random. The clerk will then be using batch processing.

Suppose the library becomes busier still, and it becomes necessary to employ two loan clerks. They may return to the direct method of recording loans, but since they share the use of the same index cards, they will need to cooperate in using them, and will sometimes even contend for the same book or user record. Given enough work, they will again find it easier to record loans and update the files separately. There are several ways they can do this. One option is for one clerk to control the book file, and the second clerk to control the branch library file. This division of work ensures that they will never have to contend for access to record cards.

At this stage, we may represent the library system using the data-flow diagram of Figure 1.4.1. From left to right, Figure 1.4.1 shows an external entity, the library users, who create 'loan requests'; a process that checks and updates the book records, creating a stream of 'approved loans'; and a process that updates the user records. Such data-flow diagrams have long been used by systems analysts to describe system designs and explore alternatives [DeMarco 1978]. The diagram is a 'logical' one; it makes no reference to its physical implementation. The form in which records are stored could be file cards or a computer database, and the processes could be enacted by humans, by computer, or a mixture of both. Data-flow diagrams leave these choices open. Figure 1.4.1 also allows the two processes to be carried out by two different clerks, or by the same clerk. For that matter, it does not specify whether the system is batch or on-line. However, the data-flow diagram records one important fact: neither set of records can be updated until the number of books in stock has been checked.

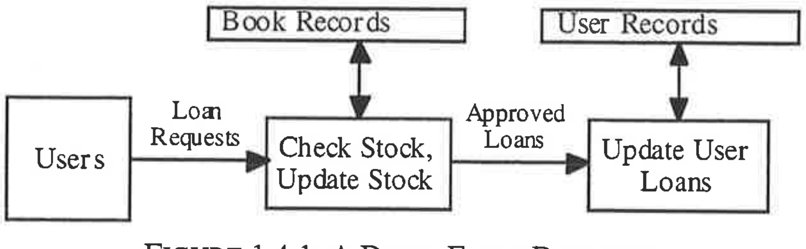

FIGURE 1.4.1: A DATA-FLOW DIAGRAM

Suppose that the library becomes so busy that more than two clerks are needed. How can they be used most efficiently?

One option is for the new clerks to record loans, while the existing clerks update the book file and branch library file. But what if the clerk updating the books file cannot cope? If two clerks are allocated to updating it, they will need to share its use, and it may prove that two clerks can work no more quickly than one. But suppose that the books file is split, A-N, O-Z. Then the two clerks can work independently, at double the speed. The same trick can be used to speed up access to the branch library file, and by splitting the files into more and more parts, the speed-up can be increased as much as needed.

There is an important principle at work here: as far as the information system is concerned, the actions on each branch library card are independent of those on every other branch library card, and the actions on each book card are independent of those on every other book card. It is this property of *independence* that allows the card files to be split into parts that can be processed in parallel — the same condition that allowed the files to be processed in A to Z order. However, the independence property could only be exploited after the system was decomposed into two steps, which relied on the property of *separability*. This example illustrates the importance of the two properties in system design.

The reader might think that it is impossible to design such a simple system badly. However, consider the data-flow diagram of Figure 1.4.2, in which updating the book records has been separated from checking them. The resulting system will work correctly only for certain implementations. For example, it will work as planned if each loan is processed completely before the next loan is started on. It will also work correctly if loans are processed in parallel, provided that no two loans can be in progress for the same book. However, it cannot work correctly as a batch system. Because all updates to 'Stock' take place after all inspections of 'Stock', if there are several requests to borrow the same book, and only two copies remain in stock, all the loans will erroneously receive approval. In the update process, the number of copies of the book may become zero or negative. The model would not correspond to a valid real-world situation.

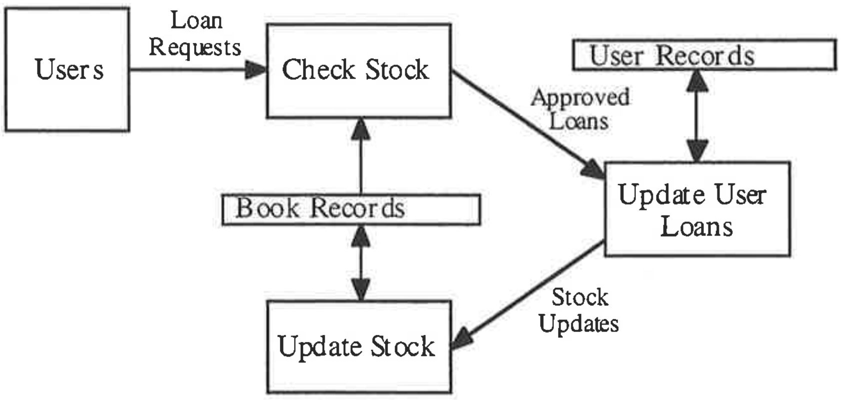

FIGURE 1.4.2: A SECOND DATA-FLOW DIAGRAM

Currently, the design of such batch systems relies on insight and experience. The dominant methodologies, SASS [DeMarco 1978] and JSD [Jackson 1983], both use the data-flow diagram as a tool. SASS (Structured Analysis and System Specification) begins by analysing the existing system. Thus, SASS might discover Figure 1.4.1 by examining a system in which there were already two clerks in operation. JSD (Jackson System Development) begins by considering the communications needed between the objects modelled by the system, which it calls 'entities'. It is essentially an object-oriented design method. Both approaches recognise that system design is a matter of transforming a simple but inefficient design into an efficient but more complex one, but they offer little advice about which transformations should be chosen, other than by appeal to experience and common sense.

In contrast, the objective of this thesis is to show that both the architecture and detailed design of an efficient batch system can be derived rigorously from its specification. Not only that, but the central design problem is computationally tractable. This design process yields a canonical decomposition of a proposed system into its smallest possible processes. The canonical decomposition is usually close to the optimum. In the case of the Macrotopian Reference Library, Figure 1.4.1 represents the canonical decomposition.

A design may be optimised with respect to a suitable cost function by combining pairs of processes derived in the canonical decomposition. The nature of the cost function will depend on external factors, for example, whether it is important to optimise throughput or response time. In theory, the optimisation problem is computationally intractable. In practice, it is small in scale and quickly solved by heuristics. For the Macrotopian Reference Library, assuming that it does not have to deal with users interactively, Figure 1.4.1 is the design that optimises throughput. The only alternative design is to combine its two processes into one, which will usually optimise response time.

In what follows, we refer to the method presented here as the 'Canonical Decomposition Method', or 'CDM' for short. The method was first outlined in an earlier report [Dwyer 1992], and a more up-to-date version appeared in [Dwyer 1998].

1.5 Modelling the Library

From now on, this thesis will use the word 'event' in preference to 'transaction'. In database theory, 'transactions' are categorised as updates, insertions or deletions. Updates change the value of an existing record, insertions create new records, and deletions remove records. Queries, which merely inspect the database, are not usually called 'transactions'. Here, an 'event' is *any* action on the database: an update, insertion, deletion or query, or a complex procedure that is any mixture of them. In a database context, 'transaction' has a second meaning: it is the unit of work that can be committed or rolled back, and again, this is not always appropriate here. In a business context, a 'transaction' often has the meaning of an exchange or contract, but this connotation is not always appropriate either. The word 'event' reminds the reader that it models a real-world occurrence. The word ‘transaction’ will be reserved for the unit of work committed in a database, or for the business sense.

Consider the Ada program fragment of Example 1.5.1. (It is assumed that the reader is reasonably familiar with Ada [Barnes 1989]. Chapter 2 describes a system specification language whose syntax is based on Ada.) Each iteration of its loop models an event ‘e’ in which some user ‘u’ attempts to borrow a copy of book with title ‘t’ from the library. ‘A(e)’ and ‘B(e)’ supply the values of ‘t’ and ‘u’ for event ‘e’. The number of books drawn by the user ‘D(u)’ is increased by one for each book borrowed, and the number of copies available in the library is decreased by one. The algorithm checks that the number of copies of the book held in the library ‘C(t)’ is initially two or more, but there is no limit on the number of books a user may borrow.

```
for e in min .. max loop
   t := A(e);
   u := B(e);
   if C(t) > 1 then
      D(u) := D(u) + 1;
      C(t) := C(t) – 1;
   end if;
end loop;
```

EXAMPLE 1.5.1: A PROGRAM FRAGMENT

This algorithm may be implemented as it stands, but it can also be implemented indirectly by three separate processes connected by two streams of data, as shown in Figure 1.5.1. The first process models the actions of the library's branches. It loops through the values of ‘e’ and, by reference to ‘A’ and ‘B’, generates a stream of (t,u) pairs, which it passes to the second process. The second process models the stock update operation. For each (t,u) pair in its input stream, it checks the value of ‘C(t)’ and conditionally decrements it, passing the value of ‘u’ to the third process. The third process models the user library loans update operation. For each ‘u’ in its input stream, it increments ‘D(u)’ once.

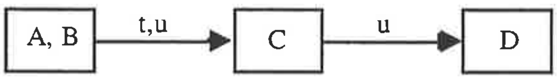

FIGURE 1.5.1: A PROCESS PIPELINE

A merit of this indirect approach to implementing the program fragment is its flexibility in scheduling the three processes. First, the processes could be executed by different physical processors, perhaps remote from each other, the data streams passing through a communication network. Second, the processes could run on the same processor on three different days of the week, with the data streams being stored in intermediate files. Third, they could run as concurrent tasks on a single processor, communicating by some kind of rendezvous. Fourth, they could be three procedures of a single program, with the data streams comprising the parameters passed at successive calls. These four options certainly do not exhaust the possibilities. A similar flexibility was described by Conway in connection with the use of coroutines [Conway 1963]. Following Conway's terminology, we call the three processes 'separable'.

As will be shown later, although it is permitted for one process to pass another a *copy* of an attribute, a database attribute may never be accessed *directly* by more than one process. Consequently, we may name a process by the set of attributes it accesses. For example, a process that accesses attributes 'A' and 'B' will typically be called '{A, B}'.

Separating the algorithm into these three processes allows further decompositions. In the second process, the values of each element 'C(t)' of 'C' depend on 'A' and 'B', but are independent of each other, so that the '{C}' process of Figure 1.5.1 actually decomposes into many smaller processes, one for each of its elements. If the domain of 'C' is large, this allows massive parallelism. Similarly, each element of 'D' can be assigned to a separate process. The resulting set of processes is suggested by Figure 1.5.2.

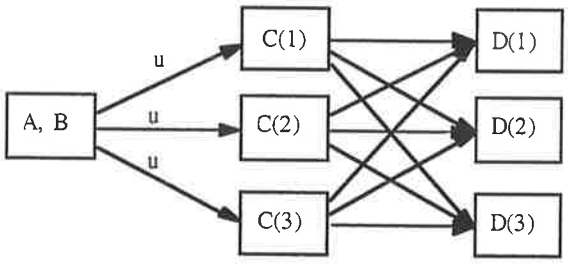

FIGURE 1.5.2: EXPLOITING PARALLEL CONCURRENCY

To accurately model the real world situation, the order of the loan events must be respected. With the suggested interpretation, if two users try to borrow the last available copy of a book, the earlier event should succeed, but the later one should fail. Consequently, although many '{C}' and '{D}' processes can operate concurrently, the system must preserve the order of the messages that reach them.

Apart from all its processes being independent of one another, Figure 1.5.2 shows a second condition that must be satisfied to exploit concurrency — the sending process must know which process of the receiving set should be sent its results. For example, because the '{A,B}' process knows the value of 't', it can direct a message (containing 'u') to the correct '{C}' process. The '{C}' process can then use the value of 'u' it received to direct a message to the correct '{D}' process.

Figure 1.5.2 shows the decomposition of the Macrotopian Reference Library system into its greatest possible number of parts. All the scenarios that were described for organising the work of the library clerks can be derived from this one diagram by grouping its processes together in different ways.

Figure 1.5.2 also suggests why the algorithm of Example 1.5.1 cannot be executed in parallel without first decomposing it. If concurrency were attempted with respect to the elements of 'C', the proposed set of parallel '{C}' processes would contend in their accesses to shared elements of 'D'. Equally, a proposed set of parallel '{D}' processes would contend in their accesses to shared elements of 'C'. A third alternative is possible when messages arrive from many sources, when each source can be associated with its own parallel process. A fourth is for events to be allocated to processes on a random, or a round-robin basis, say. These latter two alternatives would contend in the accesses to both 'C' and 'D'.

Consider again the system involving clerks and file cards. If work was divided between the clerks purely on the basis of titles, they would contend for access to user cards. If it was divided on the basis of user libraries, they would contend for access to title cards. If it was divided on the basis of source, or at random, they would contend for both sets of cards.

Such contention is resolved in database management systems by locking records while they are in use. When a transaction completes, it frees the records it has locked. A transaction that needs a record locked by another must wait until it is free again. In general, this approach has several attendant difficulties, one of which is that two transactions may deadlock: each may be waiting for the other to unlock the records it needs; neither is able to complete and release its records. Although this can happen in general, deadlock cannot happen in the library system, for the same reasons that batch processing is possible. First, because of *separability*, each transaction first locks a 'C' record, then a 'D' record, so that there cannot be a case where one has locked a 'D' record and is waiting for a 'C' record. Second, because of *independence*, each transaction locks at most one 'C' record and one 'D' record, so that there cannot be a deadlock involving two or more 'C' records or two or more 'D' records.

The '{A,B}' process cannot be executed in parallel, and must remain as a single thread. The order of its successive iterations critically affects the outcome of the algorithm and the final values of 'C' and 'D', specifically because of the control condition using 'C(t)' — and in general because the order of events is important. If a particular title is in short supply, it should be a matter of first come, first served. This would not be guaranteed if the values of 'e' were enumerated in reverse order, or processed concurrently. This puts a constraint on parallel processing in the {C} and {D} processes, in that two events affecting the same user or title should act in the correct order. The price that must be paid for this is that a process cannot act on an incoming message until it is sure that no earlier one is outstanding.

The single-thread constraint is often relaxed in practice, especially when events arrive from several geographically distributed sources. If two events are initiated at roughly the same time at two different sites, the order in which they are arrive at a central site is rather arbitrary, so it is usually considered adequate if the result of updating the database corresponds to *some* possible ordering of the events, even if it is not the order that would have been recorded by some external observer of the system. This relaxed constraint is called 'serialisability'. However, even serialisability requires that the events initiated at a given site should be completed in their correct relative order.

The basic way in which these difficulties are handled is to time-stamp events and messages. Each process must then process its incoming messages in time-stamp order rather than arrival order. If events arise at several sites, it is possible that two of them might be given the same time-stamp, in which case the site identifier may be used as a tie-breaker. In this way there does not need to be a central arbiter, which avoids a potential bottleneck. Such a system could resemble Figure 1.5.2, but would have multiple copies of the {A,B} process, one for each site.

1.6 Sequential Access

When the elements of a set can be updated independently of one another, offering the potential for parallelism, the same circumstances allow a kind of simulated parallelism. Thus, instead of there being a processor for 'C(1)', a processor for 'C(2)', etc., as in Figure 1.5.2, a single processor may support 'C(1)', 'C(2)', etc., in turn. This is sequential processing. Sequential processing might appear to offer little advantage over dealing with the elements of 'C' at random, as demanded by successive values of 't'. In practice, there is almost always some advantage to exploit.

Most computers have some form of hierarchical memory, in which data must be brought from slower storage to high speed storage to be processed. Data is usually moved in blocks or pages, of a size that typically contains many data elements. In random access processing, only one element in a block is likely to be used before the block is replaced, so that most of the block is moved uselessly; but in sequential processing, all the elements within the block are used systematically. In other words, sequential access exploits the property of memory locality. In addition, it reduces the cost of moving data even more, because it does it at most once per element.

Sequential access processes can use simpler storage structures than random access processes, for example, a linear linked list or sequential file rather than a binary search tree or an indexed file. Sequential structures are cheaper to access and update, and their support algorithms may have less computational complexity, for example, $\mathbf{O}(n)$ rather than $\mathbf{O}(n \log n)$ to access n elements. On the other hand, a sequential access process needs its input stream to be sorted, which is itself an $\mathbf{O}(e \log e)$ operation, where e is the number of input events. Despite the cost of sorting, sequential access processes often out-perform random ones.

Specifically, the advantages of sequential access are manifest in batch information systems that process large files, retrieved in blocks containing many logical records. Accessing a file sequentially means that all the records within a block can be dealt with at one time. Accessing a file randomly means that this advantage is lost. Since sequential files have a simpler internal organisation than random access files, they are also faster to update. There is therefore a premium on using sequential access whenever possible.

1.7 Scope of the Thesis

The general problem considered here is the transformation of an algorithm into several, often parallel, processes. The underlying principles have already been outlined. They are 'separability' and 'independence'. Two processes are separable if messages can be passed between them in one direction only. Two processes are independent if no messages need to be passed between them at all.

A system will be modelled by a loop that reads events, changing the state of its database and causing outputs to be written. (Although the algorithm of Example 1.5.1 creates no external output, outputs are possible in general.) The loop accessing the 'A' and 'B' arrays in Example 1.5.1 may be regarded as an abstraction of reading such an input file; the 'C' and 'D' arrays constituting the database. The system specification will always have the form of an iteration within which events update the database. Absolutely any computational requirement *can* be expressed in such an iterative form, but some requirements are easier to express than others. The theory given here is most useful for algorithms that update states in response to a stream of events unfolding over a period of time, typically referred to simply as 'systems'. Such systems arise in information processing, databases, and discrete-event simulations. A newly important area is that of 'reactive' programs, i.e., interactive programs (often with mouse-driven graphical user-interfaces) that respond to arbitrary sequences of user commands. Another emerging area of importance is 'Workflow' or 'Groupware' [Malone & Crowston 1993, Flores *et al.* 1993, Casati *et al.* 1995, Weske 1998], which allows users at several workstations to cooperate in the completion of a task by sending messages over a network. This technology is a direct automation of manual office procedures in which messages are passed as memoranda. The enabling technology provides the infrastructure, but it does not address the design issues, which are exactly those discussed here.

The decomposition of an algorithm into component processes must preserve its correctness, and it turns out that the set of possible decompositions is a property of the algorithm. Some algorithms decompose into many processes; others don't decompose at all. Generally speaking, the more component processes in the decomposition, the better the chance of finding an efficient implementation. Therefore, although a general theory of decomposition will be developed here, the theory is more useful for some kinds of problem than others. In some cases, the theory may yield trivial results. For example, a reactive system may decompose merely into a keyboard process, a computation, and a display process. The Canonical Decomposition Method is much more likely to yield interesting results in the case of an information system.

It is worth mentioning that, as the design of a product may be adapted to let it be made on a production line, so the specifications of systems are sometimes adjusted so that they yield efficient decompositions. This is an important topic. A graphical tool presented here, the 'state dependence graph', or 'SDG' provides both the basis for a mathematically rigorous design algorithm, and an intuitive understanding of system structure. An SDG helps a designer visualise the impact of the system specification on its implementation. As a result, it may often suggest ways in which a specification could be changed to lead to a better implementation. There are three ways in a which a specification can be changed: by changing the way a requirement is expressed as an algorithm, by changing the way the system is modelled, or by negotiating a change to the requirement itself.

1.8 Organisation of the Thesis

The thesis is organised in two parts. The first part concentrates mainly on the issue of 'separability', and the second on 'independence'. Thus, the first part considers the decomposition of a system into component processes, while the second discusses whether these processes can be implemented using independent (i.e., sequential or parallel) access. The first part is concerned with discovering the set of feasible designs, and the second is concerned with choosing the best of them. The reason for this division is that the conditions for independent access involve a lot more technicalities than does the division of the system into its component processes. However, the first part will make intuitive use of the ideas of the second part, by assuming that the reader can understand the conditions for independent access by the exercise of common sense. Indeed, it is one of those areas where common sense works very well, and an algorithm is harder to understand.

The first part considers the general problem of decomposing a system into separable processes. Given the system specification, it turns out that its decomposition into the maximum possible number of separable processes is canonical. Its 'canonical process graph', or CPG, is a network of loosely connected processes that allows the greatest possible concurrency whilst preserving correctness. The graph is a function of the specification.

This satisfying result is marred by two circumstances, which will be referred to as the 'specification problem' and the 'composition problem'.

There are often many ways to specify a given system requirement; some may lead to efficient decompositions, others may not. The act of formalising a requirement uniquely determines its CPG, which directly affects the efficiency of its implementation. This thesis provides a formal means of discovering the CPG. Without it, it would be hard to predict how a change to a specification would impact on efficiency — indeed, trivial changes sometimes have enormous effects. This is called the 'specification problem'.

The 'composition problem' is as follows. Because the CPG contains the greatest possible number of processes, it maximises the cost of moving data between them. It is often better to combine some minimal processes into larger composites, thus reducing the communication cost. On the other hand, it is bad to combine processes in a way that would destroy the opportunity for parallel or sequential access. The optimal composition problem is combinatorial. Its details depend on the environment in which the system must operate, including the technology underlying its implementation. Since this optimisation problem involves testing the conditions for independent access, it will be treated informally in the first part of the thesis, and its formal discussion will be left until the second part.

There is a further difference between the material in the first and second parts. The first part develops a graphical technique that can be applied by a systems analyst. The second part formalises the technique to the point that the design can be carried out by a computer program, and describes such a program, called *Designer*. The techniques of both parts, in addition to deriving a design, are capable of deriving the specifications of the processes that make up the system. In fact, these specifications are derived by simple textual transformations of the original specification. However, no program is described here for making these transformations. Writing such a program would be straightforward, but tedious. There is no question about the feasibility of such a program. On the other hand, implementing the *Designer* program proved an essential part of the thesis, for two reasons: it checks the correctness of the theory, and it demonstrates the effectiveness of the optimisation heuristics.

Unfortunately, there are some small inconsistences between the requirements of the first and second parts of the thesis. These are mainly due to conflicts between what can be done *in principle* and what has been done *in practice*. For example, an SDG can suggest an efficient process network, but the *Designer* program may be unable to derive its implementation. This is because the process specifications would need to be derived by some means that the program does not know how to do. These conflicts are felt most strongly between Chapter 5, which discusses independence informally, and Chapter 8, which describes how it may be uncovered by formal analysis. There is also some conflict between Chapter 2, which describes a specification language, and Chapter 10, which describes the subset of it that is recognised by the *Designer* program. To minimise these inconsistencies, sometimes the reader will encounter arbitrary-seeming restrictions. They are pointed out in the text.

The first part comprises Chapters 2, 3, 4, 5 and 6. The second part comprises Chapters 7, 8, 9 and 10. Finally, Chapter 11 compares the Canonical Decomposition Method (CDM) with some existing methodologies, and presents the claims of the thesis, some conclusions, and some speculations.

Chapter 2 discusses the 'event-state model' of systems, which is the basis of the system specification language used in CDM. It shows how a system specification may be transformed into an equivalent set of process specifications, using the notion of 'delayed procedure call'.

Chapter 3 reviews the means by which batch systems are implemented. It gives examples of two Cobol file update algorithms, one using random access and the other using sequential access. It also gives an example of a parallel update algorithm. It argues that the sequential and parallel algorithms are both based on the same notion of 'independent access'.

Chapter 4 introduces the notion of 'real-time equivalence' that serves as the test of correctness of a system design. It also formalises the idea of 'separability', which is the property that allows a system to be decomposed into a set of processes connected by queues, called its 'process graph'. It introduces the 'state dependence graph', or SDG, from which a CPG can be derived.

Chapter 5 deals with the issue of 'independence', the property of a process that allows it to use sequential or parallel access. It introduces a semi-formal method of determining when two processes can be combined without sacrificing independence, and an informal approach to optimising the CPG.

Chapter 6 uses the theory and techniques of the preceding chapters to examine some actual design problems. Its aim is to show how specifications affect design, and conversely, how the need for efficiency can prompt changes to a specification. This concludes the first part.

Chapter 7, which opens the second part, describes how the specifications of component processes can be derived from the system specification and a process graph. The main purpose of this chapter is to examine some factors that must be considered in deriving an optimum process graph.

Chapter 8 takes a more rigorous approach to the construction of SDG's, based on use-definition graphs similar to those used in optimising compilers. It describes some important extensions to traditional use-definition analysis that can detect opportunities for using independent access.

Chapter 9 discusses the optimisation problem. It proves that it is NP-complete. However, it also presents several heuristic methods that work well in practice.

Chapter 10 describes a computer program, *Designer*, which can derive an efficient process network from a system specification, using the techniques presented in earlier chapters.

Chapter 11 relates CDM to existing systems design methodologies, speculates about possible future developments, and states the claims of the thesis.

2. System Description

[Olle *et al.* 1991] lists three perspectives that can be used to describe systems: data oriented, process oriented and behaviour oriented. Most design methodologies use more than one perspective to describe systems. Data oriented views emphasise the relationships between data objects, usually statically, as in Entity-Relationship Modelling [Chen 1976]. Process oriented views emphasise the computer or business processes that implement a system, and typically use Data Flow Diagrams [DeMarco 1978, Gane & Sarson 1979]. Behaviour oriented views emphasise the sequences of events that can occur in the operation of a system [Teisseire 1995], typically favouring the use of Petri Net models [Maiocchi 1985, Tse & Pong 1986, Tse & Pong 1989, Preuner & Schrefl 1998]. A second approach is to use time-dependent constraints [Bidoit & Amo 1995]. Behaviour may also be defined by the syntax of a language that will generate it, and the language in turn may be represented by a process or a set of interacting processes that will generate its sentences. It is therefore possible to have a behaviour oriented methodology that is described in terms of processes, for example JSD [Jackson 1978, Jackson 1981, Jackson 1983, Cameron 1983a, Cameron 1983b, Cameron 1986] or CSP [Hoare 1978, Hoare 1985], which should not be confused with a process oriented model.

A typical view presents a relationship between some objects. Most methodologies avoid unstructured verbal descriptions. Relationships may be drawn as graphs, charted as incidence matrices, or expressed algebraically. Graphs are best where a human is expected to visualise complex relationships, matrices are usually favoured when the documentation is checked or analysed by the computer, and algebraic methods are favoured in connection with formal derivation and proof.

[Olle *et al.* 1991] identify 48 distinct analysis products that can be described. Methodologies differ in which of these possible products they consider are worth describing, the perspectives they take of these objects, and the conventions they choose for documenting them. They also differ in what aspects of system development they address, for example, financial and manpower considerations, the description of existing systems, or the design of system implementations. Most methodologies aim to assist with system development, but they differ with respect to the stages that they support. Given these variables, a virtual infinity of methodologies is possible. There are several good reviews of methodologies available [Bubenko 1986, Mannino 1987].

The Canonical Decomposition Method (CDM) described here starts with a behaviour-oriented specification and derives a process-oriented design from it. To a lesser extent, the specification is also data oriented. CDM is able to refine an abstract data specification to yield a set of file descriptions.

What is special about CDM? It is a formal approach that transforms a specification into a design solution. Unlike existing methods, it needs no external help. That is, it is not a means by which a designer can derive a design from a specification, it derives one by itself. Its only help is a cost function to determine which design alternative is best.

There is little point in trying to relate CDM to other methodologies at this point. It is a radical departure from most of them. It is the thesis of CDM that the structure of systems must be derived upwards from primitives. Most existing methodologies assume that the designer can, and should, impose the gross structure on the system before filling in the details. Often, the gross structure of a new system design is derived from the gross structure of an existing system. According to CDM, this only works, if it works, because both systems have gross structures that are determined by the same primitives. We may summarise the difference between CDM and other methods by saying that CDM is bottom-up rather than top-down.

Some existing methodologies are transformational, in that a requirement is transformed into a design, usually in several stages. These methods typically suffer from two drawbacks: there is little guidance about which transformations should be made, and the transformations themselves can be complex. Determining the correct sequence of transformations is difficult because the design problem is posed as the decomposition of a system into parts, and it takes experience to guess which parts will be needed.

CDM shares this transformational property, but its transformations are very simple, being based on text substitution. The sequence of transformations is simplified too, by first decomposing a system *canonically* into the maximum possible number of parts, then combining them again. This process of canonical decomposition followed by composition both limits the search space for a solution, and provides a sense of direction in searching it. The relationship between CDM and other methodologies will be discussed in Section 11.1.

2.1 System Models

System behaviours are typically specified by models that are either state based or sequence based. A state-based model is described in terms of events, which modify the values of state variables or trigger the generation of output. Events are externally caused, and unfold as a sequence in time. A sequence-based description expresses the output sequence of the system (its behaviour) as a function of its input sequence (i.e., its history of events).

A disadvantage of a state-based model is that it is necessary to decide how to represent the system state, i.e., what set of variables should be stored. This thesis shows that the decision sometimes turns out to have a profound effect on the number of processes into which a system can be decomposed, and on the efficiency of its implementation. This issue will be referred to as 'the state representation problem', and is an aspect of the specification problem.

A sequence-based description avoids the explicit mention of states, and would seem to finesse the state representation problem. However, there are strong reasons for believing that state-based descriptions are better suited to the analysis given here — but the reasons can better be discussed after the analysis has been presented. Briefly, the state representation problem is not avoided in a sequence-based model; there are often many ways of defining the grammars of its input and output sequences, each of which would be recognised or generated by a corresponding state machine. Therefore, there are still many ways of describing the same behaviour, and each of them corresponds to an underlying state machine. Proving the equivalence of two regular grammars is intractable in general, and proving the equivalence of two context-free grammars is undecidable [Aho & Ullman 1972a, Hopcroft & Ullman 1979b]. Such a proof, when it exists, usually amounts to showing that both grammars are recognised by isomorphic state machines. So, why not deal with the state machines directly?

A state-based description of a system requires each kind of event to be specified by the outputs it generates, and its effect on the system state. The effect can be specified by an algorithm that brings about the required relationship. This choice is used here. It creates the interesting situation that the algorithm can be decomposed to derive algorithms for the component processes that make up the system. Thus it promises a means of automating system construction. Unfortunately, there are sometimes several algorithms that can satisfy the same relationship, some leading to greater concurrency than others. Selecting the best algorithm will be referred to as 'the event specification problem', and is a second aspect of the specification problem.

Alternatively, an event may be specified by the relationship between the states of the system before and after the event. It might seem that this would finesse the event specification problem. However, the problem reappears because there are sometimes many mathematically equivalent ways to formulate the same relationship. Proving the equivalence of two formulations can be undecidable, and transforming a formulation into its equivalents is intractable. It is also a reasonable assumption that, given some automatic means to turn before and after relationships into algorithms, the different ways of expressing the relationship would lead to exactly the same set of system implementations that could be obtained by specifying an event by algorithms directly.

It seems that neither the event nor state specification problems can be avoided, and although they are discussed in this thesis, it offers no solution to them.

Complex systems typically accept many kinds of input events, calling for different responses. Some events may cause the state of the system to change, others may cause it to be inspected, yet others may do both. For example, the library example of Figure 1.4.1 (Page 6) could also allow users to return books, as in Example 2.1.1, where K(e) is the kind of the event 'e'.

```
for e in min..max loop
   t := A(e);
   u := B(e);
   if K(e) = Borrow then
      if C(t) > 1 then
         D(u) := D(u) + 1;
         C(t) := C(t) – 1;
      end if;
   elsif K(e) = Reshelve then
      D(u) := D(u) – 1;
      C(t) := C(t) + 1;
   end if;
end loop;
```

EXAMPLE 2.1.1: TWO KINDS OF EVENT

Since a state-based description assumes a sequence of input events, some kind of read loop is needed to drive it. The event loop of Example 2.1.2 suggests the general form of a system.

```
begin
   loop
      "read the next input event";
      case "type of event" is
      when "event type 1" => "execute the procedure for event type 1";
      when "event type 2" => "execute the procedure for event type 2";
      ...
      when "event type n" => "execute the procedure for event type n";
      end case;
   end loop;
end;
```

EXAMPLE 2.1.2: A SYSTEM READ LOOP

```
procedure Borrow (t: title; u : user) is
begin
   if C(t) > 1 then
      D(u) := D(u) + 1;
      C(t) := C(t) – 1;
   end if;
end Borrow;
procedure Reshelve (t: title; u : user) is
begin
   D(u) := D(u) – 1;
   C(t) := C(t) + 1;
end Reshelve;
```

EXAMPLE 2.1.3: EVENT SPECIFICATIONS

Assuming this outline, it is only necessary to define the procedures that are activated for each kind of event. This being so, an event may be seen as semantically equivalent to a call of such a procedure. A system description may therefore consist of a declaration of some state variables and the procedures that can act on them. In Ada, such 'event procedures' could be written as in Example 2.1.3. (The specification language will be described in Section 2.6.)

2.2 Process Graphs

The building block of a process graph is shown in Figure 2.2.1. A rectangular box represents a system or one of its components, and is usually referred to as a 'process'. The box is labelled internally with the names of the system variables that it accesses ('A' and 'B' in Figure 2.2.1). A process may have several inputs and outputs, which are unlabelled.

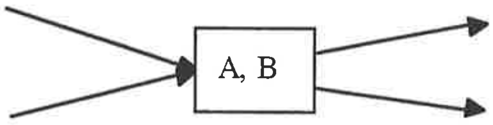

FIGURE 2.2.1: A PROCESS GRAPH

All inputs are drawn entering a process at a single point. This symbolises that they interleave to form a single sequence in time. Even if the inputs arrive from different sources, it will be assumed that they are totally ordered — by an arbiter or global clock, for example. Output streams are shown leaving the process at separate points. This symbolises that although a system is the 'victim' of its inputs, it has total control over the sequence of its outputs.

Input and output events need not occur in one-one correspondence. A given input event may produce any number of output events, including zero. The sequence of output events generated by a component process is usually ordered in time. For example, if the entire action of Figure 1.5.1 were considered as a single event, one input to the '{A,B}' process would trigger its event loop, generating an ordered stream of 't, u' input events for '{C}'.

When an input arrives, its corresponding event procedure is activated. An event procedure is assumed to be atomic, i.e., to complete its action without interference. The processing of one event must be completed before the processing of the next can begin. If a new input arrives at a process during execution of an event procedure, its action must be delayed until the procedure completes. The means of causing the delay is not given in the specification, but may be assumed to be a first-in first-out queue of some kind.

Systems exist where a total ordering of inputs is not required, and for which event-state based descriptions are therefore inappropriate. One such situation is when inputs arrive from several sources which hardly interact with one another. (If they don't interact at all, they are separate systems.) Interactions occur when events from different sources contend in their use of the database. One or other source may then be given priority only at the time of contention. For example, an event that has already acquired a record lock may be given priority over one that hasn't — in which case some means may be needed to deal with deadlocks between events. However, events that don't happen to interact have only a partial ordering; the order in which they are processed won't affect the final state of the database. Such systems are certainly handled correctly by CDM, but its criterion of correctness is stricter than that usually applied to them.

A more profound breakdown of the total ordering assumption occurs when a system is 'relativistic'. A relativistic system is one in which the time taken to respond to an event is less than the time taken for a message to propagate across the system. This can happen because the system is geographically dispersed, or because it is very fast. The global telephone network is an example of a relativistic system. Suppose you decide to call a friend who lives on the opposite side of the world, and at the same time, they decide to call you. There is a true sense in which these two things can happen at the same time, because the two parties can initiate messages whose paths cross. (This may be contrasted with Ethernet, where crossing messages are rejected and retransmitted, but is similar to electronic mail.) Both parties would believe that they had initiated the call and ought to pay for it. It is just as well for them that the actual telephone network will reject both calls. The problem with describing such a system is that the order of events depends on the observer. It is often assumed that the state of a system is simply the product of the states of its parts, but this is only the view seen by a single observer; different observers of a relativistic system may still disagree about the history of the system's state. A conventional system description forces us to specify that events have some underlying partial ordering, such that if they were applied according to that order, the correct system state would be obtained. An attempt to force any similar ordering on the telephone network would bring it to a halt. If there were a global ordering, a local call could not be connected until it was first checked that the caller had no incoming international call in transit, which would simply take too long. (For example, if messages were switched through a tree network to impose an ordering on them, about half the international calls would have to pass through a single world-wide switch.) Relativistic systems cannot be described by a simple event-state or behavioural model, and have different criteria of correctness. However, they usually contain non-relativistic parts to which the theory presented here may be applied successfully.

As a matter of terminology, the components of a system will be referred to here as 'component processes' (or simply 'processes') rather than 'sub-systems'. In information systems usage, the term 'sub-system' is used to denote a part of a system that implements only a subset of the kinds of its input events (as the term is often used in information processing). Since different sub-systems in this sense cannot operate concurrently, here they will be referred to as 'modes'.

2.3 Delayed and Local Procedure Calls

When processes '{C}' and '{D}' are connected by a data flow from '{C}' to '{D}', process '{C}' writes an output that becomes an input to '{D}'. As mentioned earlier, this flow can be implemented by a transfer file, queue, rendezvous, etc. The simplest case is a procedure call, i.e., '{C}' directly invokes a procedure in '{D}'. Irrespective of the actual implementation, the action of '{D}' can therefore be defined by a procedure — exactly like an event procedure of a system specification. This is appealing, because it makes the description of a component process just like the description of a system as a whole. Correspondingly, when process '{C}' sends a message to process '{D}', the communication may be modelled by a procedure call. The corollary is that an output from the system as a whole should be modelled by a procedure call — to a process external to the system being modelled. To complete their correspondence with event procedures, procedures in component processes are also assumed to be atomic.

A procedure call does not have to invoke its corresponding procedure immediately. The call may write a record to a transfer file or queue. When a loop within the receiving process reads the record, it will activate the invoked procedure. Thus, there may be an arbitrary delay between the call of a procedure and its activation. This model of message passing will be referred to as 'delayed procedure call' (or simply 'delayed call').

Because of the one-way nature of the data flow, only input parameters are allowed in delayed procedure calls. It would make no sense to allow output parameters. Assuming they return values to be used by the sending process, the sending process could not complete its procedure call until the called procedure in the receiving process had completed. This would forbid a delay between sending and receiving processes, defeating the object of the decomposition. Delayed procedure call is therefore similar to the 'send' primitive of Andrews's Synchronising Resources [Andrews 1981].

(Although procedures with output parameters but no input parameters would also enforce one-way data flow, they would not model files or queues so well; the call would need to be made from the receiving process to the sending process.)

This does not mean that event procedures with output parameters are not permitted. It means that calls to them cannot be delayed. Such event procedures can be used to model processes that may be physically dispersed, but which must synchronise their activities. Such inter-process communications will be referred to as 'remote procedure calls' — whether the processes involved are physically remote or not. These are similar to Andrews's 'call' primitive.

The event specifications of Example 2.1.3 can be decomposed as given in Example 2.3.1. The first two procedures correspond to the 'C' process of Figure 1.5.1; the second two procedures correspond to 'D'. Even though the call of 'Increase' is not the final statement of 'Borrow', 'Borrow' may continue to its completion as soon as it has made the call. This is precisely because 'Increase' has no output parameters. A similar relationship holds between 'Reshelve' and 'Decrease'.

The semantics of any procedure call may be discovered by replacing the call by the body of the called procedure, substituting actual parameters for formal parameters. (There is a trivial substitution of 'u' for 'u' in this example.) If this is done, Example 2.3.1 can be seen to be equivalent to Example 2.1.3.

```
-- C
procedure Borrow (t: title; u : user) is
begin
   if C(t) > 1 then
      Increase(u);
      C(t) := C(t) – 1;
   end if;
end Borrow;
procedure Reshelve (t: title; u : user) is
begin
   Decrease(u);
   C(t) := C(t) + 1;
end Reshelve;
-- D
procedure Increase (u : user) is
begin
   D(u) := D(u) + 1;
end Increase;
procedure Decrease (u : user) is
begin
   D(u) := D(u) – 1;
end Decrease;
```

EXAMPLE 2.3.1: COMPONENT PROCESS PROCEDURES

2.4 Data Structures

An abstract view is taken of data structures, and only one method of forming structures is allowed, the functional dependency, or 'FD'. The notion of 'functional dependency' used here is exactly that used in relational database theory [Codd 1970], from which it is well-known that it is universal enough to model any data relationship [Grant 1987]. It is also the basis of at least one data modelling technique [Frankel 1979, Bertziss 1986].

FD's will be *modelled by* arrays. Array notation provides a convenient way to refer to elements of sets, such as 'C(t)' or 'D(u)' in the above examples. However, arrays are only a model. Typical user identifiers or titles have far too many values to be the basis of arrays that could fit into a computer's primary storage. In most situations it would be necessary to use files or database tables to store the volume of data involved. Indeed, the numbers of elements needed by the array model are not merely the numbers of *actual* users or titles, but the numbers of *possible* users or titles. (In this thesis, the terms 'files' and 'tables' and the terms 'records' and 'rows' are used almost interchangeably.)

An FD models the mathematical notion of a function from a source domain to a target codomain (or range). A function may be either total or partial. A total function has a codomain value corresponding to every argument in its domain. A partial function has values for particular arguments only. A total function may be modelled by an array with a default value; each element of the array is initialised to the default. For every argument, the value of the FD is initially the default value, and its value always remains defined. A partial function may be modelled by an uninitialised array. Initially, its value for every argument is undefined. Only those elements that have been assigned values are meaningful. Whether the value of a particular argument is defined or not is determined by context, for example, the function may be defined only for arguments that are known to have some other property, for example, to be members of a particular set.

Following Ada syntax, the FD's of the library system would be declared as follows:

C : **array** (title) **of** natural := (**others** => 0);
D : **array** (user) **of** natural := (**others** => 0);

where the initialisations ensure that there are no books either in the library or on loan when the system is created.

Although a set is not an FD, a set may be modelled by its characteristic function, an FD whose codomain is the boolean set {true, false}. Arguments yielding the value 'true' are considered to be members of the set, and those that yield 'false' are excluded. An empty set is modelled by an array with all elements false.

There are many ways in which a functional dependency can be represented physically. As a concrete model, the reader may imagine that they are implemented by files or database tables. Each row has an argument as its primary key and the corresponding value as its sole non-key attribute. Specifying the argument therefore enables the matching row to be retrieved, and the corresponding value to be read or updated.

Even using an external file or database table, it is not usually practical to store a row for every argument in the domain of a typical FD. Arguments that yield default values need not be stored. Therefore, if an argument has no matching row in the table, the value of the FD is its default value. This means that the initial state of the library can be represented by two empty tables. The convention also allows rows containing default values to be deleted from the database automatically. In turn, this suggests that it is better for partial functions to have default values too, to allow the storage they use to be reclaimed. In this respect, FD's are similar to sparse arrays, i.e., arrays in which only non-default values are stored.

To save space, where there are several FD's with the same domain, they may be allowed to share the same table. Specifying an argument would retrieve a row containing the values of all the FD's in the table. If, as is often the case, more than one FD was used by an algorithm, this would allow several references to be implemented by a single record retrieval. But whether two or more FD's *ought* to share one table or be in separate tables proves to be a design issue that is an output of the Canonical Decomposition Method. In specifications, all FD's will be modelled as having separate representations; there are no record structures.

Using default values means that there is no need to distinguish insertion and deletion of records from other kinds of update. In each case the value of an attribute is changed: insertion is the special case when the old value is the default value, deletion is the case when the new value is the default value. ‘Missing’ records are not exceptions; they are merely assumed to have default values.

These conveniences of the array model must be weighed against two minor inconveniences. The first is that it is impossible to update the key of a record; the contents of the array element with the old key must be moved to the element with the new key, corresponding to deleting one row and inserting another. (Such key updates must always be physically implemented by deletion and insertion anyway.) The second is that testing for the presence or absence of a given row in a conventional database table reveals a hidden boolean attribute. Such hidden attributes must be defined explicitly in this model; it is often necessary to declare a boolean array that specifies, for example, which user identifiers are allocated to actual users.

Two general assumptions are made about any implementation of FD’s. First, given a random argument of the domain, the corresponding row of the table can be read or updated in less time than it would take to read or update the whole table. Second, it is quicker to access all the (non-default) rows of an FD using parallel or serial processing than it is to access them all at random. Broadly, the implementation is assumed to be like an indexed-sequential file.

2.5 A Specification Example

Rather than invent a completely novel specification language, the thesis will present specifications in a variant of Ada [Barnes 1989]. The main changes to Ada are these:

- The only data structures are arrays that model FD’s.
- The basis of an array may be any simple type, or any list of simple types.
- A parallel loop construct is provided.
- Arrays of packages are allowed.

A system will be represented by an Ada package, whose external procedures represent the events to which it responds.

To illustrate the general flavour of a specification, Example 2.5.1 specifies the Macrotopian library system. In addition to the ‘Borrow’ and ‘Reshelve’ event procedures given in earlier examples, a ‘Buy’ event has been included so that the database can be populated, and an ‘Audit’ event serves as an example of a global query. The ‘Audit’ event counts the numbers of books in stock and on loan, and passes these numbers to a ‘Report’ package for presentation to a user. (This is an example of the system generating an external output.) Several more event specifications would still be needed to make the library model realistic.

```
with Report;
generic
   type user is private;
   type title is private;
package Library is
   procedure Buy (t: title);
   procedure Borrow (t: title; u : user);
   procedure Reshelve (t: title; u : user);
   procedure Audit;
end Library;

package body Library is
   C : array (title) of natural := (others => 0);
   D : array (user) of natural := (others => 0);
   procedure Buy (t: title) is
   begin
      C(t) := C(t) + 1;
   end Buy;
   procedure Borrow (t: title; u : user) is
   begin
      if C(t) > 1 then
         D(u) := D(u) + 1;
         C(t) := C(t) – 1;
      end if;
   end Borrow;
   procedure Reshelve (t: title; u : user) is
   begin
      D(u) := D(u) – 1;
      C(t) := C(t) + 1;
   end Reshelve;
   procedure Audit is
      Loans, Stock: natural := 0;
   begin
      all t in title loop
         Stock := Stock + C(t);
      end loop;
      all u in user loop
         Loans := Loans + D(u);
      end loop;
      Report.Audit (Loans, Stock);
   end Audit;
end Library;
```

EXAMPLE 2.5.1: A SIMPLE LIBRARY SYSTEM

Packages may also be used to describe the decomposition of a system into its component processes. Example 2.5.2a shows the decomposition of the library system into components ‘C’ and ‘D’. The procedural part of Example 2.5.2a is trivial, serving only to forward the events reaching the system to ‘C’. The ‘C’ and ‘D’ arrays are no longer global to the system, but local to its components.

```
package body Library is
   package C is
      procedure Buy (t: title);
      procedure Borrow (t: title; u : user);
      procedure Reshelve (t: title; u : user);
      procedure Audit;
   end C;
   package D is
      procedure Borrow (u : user);
      procedure Reshelve (u : user);
      procedure Audit (Stock : natural);
   end D;
   procedure Buy (t: title) is
   begin
      C.Buy(t);
   end Buy;
   procedure Borrow (t: title; u : user) is
   begin
      C.Borrow(t, u);
   end Borrow;
   procedure Reshelve (t: title; u : user) is
   begin
      C.Reshelve(t, u);
   end Reshelve;
   procedure Audit is
   begin
      C.Audit;
   end Audit;
end Library;
```

EXAMPLE 2.5.2A: LIBRARY COMPONENT SPECIFICATIONS (PART 1)

Packages may also describe the components themselves, as shown in the package body of Example 2.5.2b, which describes 'C' of Figure 1.5.1. The 'C' package contains the 'C' array. Delayed procedure call is used to model communication between the 'C' and 'D' processes. ('D' performs no action for 'Buy' events.)

```
package body C is
   C : array (title) of natural := (others => 0);
   procedure Buy (t: title) is
   begin
      C(t) := C(t) + 1;
   end Buy;
   procedure Borrow (t: title; u : user) is
   begin
      if C(t) > 1 then
         D.Borrow(u);
         C(t) := C(t) – 1;
      end if;
   end Borrow;
   procedure Reshelve (t: title; u : user) is
   begin
      D.Reshelve (u);
      C(t) := C(t) + 1;
   end Reshelve;
   procedure Audit is
      Stock: natural := 0;
   begin
      all t in title loop
         Stock := Stock + C(t);
      end loop;
      D.Audit (Stock);
   end Audit;
end C;
```

EXAMPLE 2.5.2B: LIBRARY COMPONENT SPECIFICATIONS (PART 2)

Finally, Example 2.5.2c shows the structure of the 'D' component. It contains the 'D' array. The procedure call on the 'Report' package is assumed to cause the values of 'Loans' and 'Stock' to be displayed or reported in some way.

```
package body D is
   D : array (user) of natural := (others => 0);
   procedure Borrow (u : user) is
   begin
      D(u) := D(u) + 1;
   end Borrow;
   procedure Reshelve (u : user) is
   begin
      D(u) := D(u) – 1;
   end Reshelve;
   procedure Audit (Stock : natural) is
      Loans: natural := 0;
   begin
      all u in user loop
         Loans := Loans + D(u);
      end loop;
      Report.Audit (Loans, Stock);
   end Audit;
end D;
```

EXAMPLE 2.5.2C: LIBRARY COMPONENT SPECIFICATIONS (PART 3)

```
package body C is
   array (title) of package C_title is
      procedure Buy;
      procedure Borrow (u : user);
      procedure Reshelve (u : user);
      procedure Audit;
   end C_title;
   package body C_title is
      C : natural := 0;
      procedure Buy is
      begin
         C := C + 1;
      end Buy;
      procedure Borrow (u : user) is
      begin
         if C > 1 then
            C := C – 1;
            D.Borrow (u);
         end if;
      end Borrow;
      procedure Reshelve (u : user) is
      begin
         C := C + 1;
         D.Reshelve (u);
      end Reshelve;
      procedure Audit (C1: out natural) is
      begin
         C1 := C;
      end Audit;
   end C_title;
   procedure Buy (t : title) is
   begin
      C_title (t).Buy;
   end Buy;
   procedure Borrow (t : title; u : user) is
   begin
      C_title (t).Borrow (u);
   end Borrow;
   procedure Reshelve (t : title; u : user) is
   begin
      C_title (t).Reshelve (u);
   end Reshelve;
   procedure Audit is
      Stock, C1 : natural := 0;
   begin
      all t in title loop
         C_title(t).Audit(C1);
         Stock := Stock + C1;
      end loop;
      D.Audit (Stock);
   end Audit;
end C;
```

EXAMPLE 2.5.3A: EXPRESSING INDEPENDENT ACCESS (PART 1)

As an elaboration of the design, Example 2.5.3a shows how the ‘C’ process can be further factorised into many independent processes. In Example 2.5.3a there are as many ‘C_title’ components as titles. For any realistic range of identifiers, the number of component processes would easily exceed the number of physical processors in present day parallel computers. Each physical processor must therefore support many component processes, whose specifications are given by 'C_title'. However, only those processes representing real books or real users would ever be activated. Alternatively Example 2.5.3a can be seen as a model of sequential file access, with each 'C_title' process representing the actions for each master file record, and the 'C' shell handling file-level activity.

In Example 2.5.3a, a procedure call of the form 'C_title(t).Borrow(u)' means that a call is made to procedure 'Borrow' in instance 't' of component 'C_title', passing 'u' as a parameter. However, the call 'C_title(t).Audit(C1)' cannot be implemented as a delayed procedure call, because it returns an output parameter.

Example 2.5.3a is an approximate model of both sequential and parallel access. The 'Stock' variable, of which there is a separate instance for each 'Audit' event, is updated within the 'C' process shell itself. In a parallel implementation, this is the only way to ensure that the updates to 'Stock' are properly coordinated. In other words, 'Stock' belongs to the 'C' process, and can only be accessed by it. The 'C_title' processes return the values of 'C' to be added to 'Stock'. Example 2.5.3a is only a rough abstraction of a parallel algorithm. To model parallel access more accurately is more tedious than enlightening. An efficient implementation must avoid the bottleneck of passing procedure calls to 'D' and pass them directly to its internal parallel components. Section 3.5 explains how this is done.

Example 2.5.3b shows a similar decomposition of the 'D' process, there being one instance of 'D_user' for each user.

```
package body D is
   array (user) of package D_user is
      procedure Borrow (u : user);
      procedure Reshelve (u : user);
      procedure Audit (Stock : natural);
   end D_user;
   package body D_user is
      D : natural := 0;
      procedure Borrow is
      begin
         D := D + 1;
      end Borrow;
      procedure Reshelve is
      begin
         D := D – 1;
      end Reshelve;
      procedure Audit (D1: out natural) is
      begin
         D1 := D;
      end Audit;
   end D_user;
   procedure Borrow (u : user) is
   begin
      D_user (u).Borrow;
   end Borrow;
   procedure Reshelve (u : user) is
   begin
      D_user (u).Reshelve;
   end Reshelve;
   procedure Audit (Stock : natural) is
      D1, Loans : natural := 0;
   begin
      all u in user loop
         D_user (u).Audit (D1);
         Loans := Loans + D1;
      end loop;
      Report.Audit (Stock, Loans);
   end Audit;
end D;
```

EXAMPLE 2.5.3B: EXPRESSING INDEPENDENT ACCESS (PART 2)

2.6 The Specification Language

The specification language is based on Ada package specifications. We make a distinction between two dialects of the language. The first, described below, is used both to define systems and to describe their implementations. It allows implementations to be derived from specifications using rewriting rules. The second dialect, which is a subset of the first, defines those specifications that are acceptable to the *Designer* program, and is described in Chapter 10.

The form of a system specification is as follows:

system_specification ::=
    package_specification
    package_body
package_specification ::=
    [*external_package_declarations*]
    [*generic_type_declarations*]
    package *system_name* **is**
        { *event_specification* }
    end *system_name* **;**
package_body ::=
    package body *system_name* **is**
        { *component_specification* }
        { *component_definition* }
        { *state_declaration* }
        { *procedure_definition* | *function_definition* }
        { *event_definition* }
    end *system_name* **;**

In describing syntax, most items in italics represent non-terminal symbols of the grammar, which are defined by one or more grammar rules. The exceptions are 'identifier' and 'constant', which are defined as in Ada. Words and special characters in bold type denote themselves. Where an expression appears in braces, i.e., '{...}', it may be repeated any number of times, including zero. Where an expression appears in brackets, i.e., '[...]', it may either appear once, or not at all. A vertical stroke indicates a choice between alternatives.

2.6.1 External Interface

A system is defined by its external interface, given by the package specification; and by its implementation, given by the package body. It is possible to separate a package specification from its body. In particular, when different implementations of a system are discussed, the package specification will be given once, and alternative versions of its body will be shown.

external_package_declarations ::=
    with *system_name* { **,** *system_name* } **;**

A package specification may import externally defined packages and data types. External package specifications serve two purposes. Primarily, they introduce the names of external system components to which output may be sent. As such, they usually represent processes that format output for display or printing. Alternatively, they can represent other systems within the same environment. In either case, they provide a convenient way to avoid specifying the details of output operations. Thus, if the clause '**with** Report' introduces the name of an external package, then 'Report' is assumed to contain a 'Loan' procedure, which 'Report.Loan (t,u)' invokes. 'Report.Loan' may be assumed to display or print 't' and 'u', neatly formatted. In truth, it does not matter what 'Report.Loan' does. From the viewpoint of the system being specified, 'Report' is simply an information sink. A secondary purpose of external package specifications is to support data types. For example, it would be possible to specify the clause '**with** money' to support arithmetic operations on a 'money' data type.

In addition to any data types introduced through external packages, it is assumed that the data types and operators found in the Ada 'standard' package are always available. These include 'integer', 'natural', 'positive', 'float', 'boolean', 'character', and 'string'.

generic_type_declarations ::=
        generic {**type** *type_name* **is private;** | **type** *type_name* **is range <> ;**}

Generic type declarations may be used to introduce data types whose precise form does not affect the system specification process. A **private** data type may be used in assignments, tested for equality, used as the basis of an array, or the domain of a loop variable. Thus, introducing 'book' as a **private** data type allows variables of type 'book' to be declared, and so on. **Private** types have arbitrary ordering, so that it is meaningless to ask which of two 'books' is smaller, for example, or to perform arithmetic on book identifiers. **Range** data types serve similar purposes, but also allow integer arithmetic and comparison. If more is required of a data type, it must be introduced by an external package.

event_specification ::=
        procedure *event_name* [(*event_parameters* {; {*event_parameters* })] ;
event_parameters := *parameter_name* { , *parameter_name* } : *type_name*

Event specifications define the system interface to which external input can be sent. Abstractly, each input has the form of a procedure call, naming an event that appears in an event specification, and providing a corresponding set of parameters. It is assumed that inputs sent to the entire system are totally ordered in time — not merely those sent to each system interface. An event having only input parameters may be called using a delayed procedure call. An event having output parameters must be called using a remote procedure call. Only delayed procedure call allows the called process to lag arbitrarily behind the calling process.

component_specification ::=
        [**array** (*type_name* { , *type_name* }) **of**]
        package *component_name* **is**
            { *event_specification* }
        end *component_name* ;

When a system comprises several components, each component is specified by a package specification similar to that used to define the system as a whole. Component specifications have no external package declarations or generic type declarations of their own, but inherit the declarations of their parent system. Arrays of component packages may be declared. There is a separate instance of the package for each value of its array basis. Arrays of packages are used to model sets of parallel processes — either real, or simulated by sequential access.

```
component_definition ::=
        package body component_name is
            { component_specification }
            { component_definition }
            { state_variable_declaration }
            { procedure_definition | function_definition }
            { event_definition }
        end component_name ;
```

The implementation of a system or a system component is defined by a package body. Component definitions are nested within the system definition, and it is possible to nest component definitions within other component definitions. A component may receive input from another component, or from an event procedure of the enclosing system. It may direct output to another component or to an external package. It is not possible for a system component to send output to itself (i.e., call one of its own event procedures), either directly or indirectly. However, an event procedure may call an *internal* procedure or function, i.e., one defined within the same package body but not declared in the package specification.

2.6.2 Variables

```
state_variable_declaration ::= variable_declaration
variable_declaration ::= variable_name {, state_name }: type_name := constant ;
variable_declaration ::= variable_name { , state_name } :
        array ( type_name {, type_name }) of type_name := initial_value ;
initial_value ::= ( others => constant | others => initial_value )
```

State variables may be either simple, or arrays having one or more dimensions. Each state variable must be uniformly initialised to a default value. The default must be a constant, typically 'false', '0', or the empty string. Generic **private** types must be initialised to the special value **null**. Integer, boolean, float, character and string constants are also allowed, as in Ada.

State variables are private to their immediately enclosing components. They may be read or updated by any procedure defined within the component that declares them, but not by any other component, even an enclosing component.

Local variables may be declared within any function or procedure. Local variables are declared in the same way as state variables. Local variables are always initialised on entry to the function or procedure, and discarded on exit.

A **private** data type may be the basis of an array, even though this not legal in Ada. However, the actual implementation of an array is supposed to use storage only in proportion to the number of its entries that have non-default values. It is understood that even though its basis may have infinitely many values (e.g., integer), its physical storage size stays bounded, and it is practical to implement it as a database table, for example. Similarly, if its entries are reset to the default value, the space they occupy may be reclaimed.

2.6.3 Procedures

```
procedure_definition ::=
        procedure procedure_name
            [( procedure_formal_parameters {; { procedure_formal_parameters })] is
            { local_variable_declaration }
            { procedure_definition | function_definition }
        begin
            statement_list
        end procedure_name ;
procedure_formal_parameters :=
                parameter_name {, parameter_name } : [in] [out] type_name
local_variable_declaration ::= variable_declaration
function_definition ::=
        function function_name (function_parameters {; { function_parameters })
            return type_name is
            { local_variable_declaration }
            { function_definition }
        begin
            statement_list
        end function_name ;
function_parameters := parameter_name {, parameter_name } : type_name
```

There are two kinds of procedures: event procedures, and internal procedures. An event procedure must be called from outside the package where it is declared. An internal procedure may be called by an event procedure or another internal procedure within the same package. An internal procedure defined within a component package may be called by any procedure within the component package itself. An internal procedure defined within a system package may be called by any procedure within the system package, but not within one of its components. Internal procedures may be used to encapsulate the common parts of similar event procedures. Internal functions have a similar purpose. They may have only input parameters, and may not refer to non-local variables. (They are therefore 'purer' than in Ada.) A procedure may refer to state variables within its enclosing package. A procedure may not refer to a state variable or non-local variable both directly and as a parameter.

An event procedure differs from an internal procedure in two respects: it may only be called from a different package, and it may not be nested within another procedure. Event procedures are textually distinguished from internal procedures by being declared in the system or component package specification. Internal procedures may have output parameters, but event procedures may not.

2.6.4 Statements

```
statement_list ::= statement ; { statement ; } | null ;
statement ::= assignment procedure_call | conditional_statement |
        for_loop | all_loop | while_loop
assignment ::= variable := expression
variable ::= variable_name [ ( index_name {, index_name }) ]
index_name := variable_name
expression ::= term { operator term }
```

term ::= *variable* | *constant* | (*expression*) | *function_call* | *operator term*
function_call ::= *function_name* (*actual_parameter* {, *actual_parameter* })
actual_parameter ::= *expression*

Assignment is the primary means of changing the system state. It is semantically and syntactically similar to assignment in Ada. After an assignment, its left-side variable has the value its right-side expression had before the assignment. Expressions follow the same general form as in Ada. An operator may be any of the standard operators of Ada.

A variable may be simple, or it may be indexed by one or more indices. An index must be a simple variable, not a general expression. This restriction is made to simplify later discussion. It is not a limitation on what can be specified, because any desired index expression can be assigned to a local variable that is then used as an index.

```
procedure_call ::=
        [ package_name [ ( variable_name { , variable_name } ) ] . ]
        procedure_name [ ( actual_parameter { , actual_parameter } ) ]
```

A procedure call may invoke an event procedure declared within a different package. If the called package is specified with the 'array' option, the package name must be indexed, so that a particular instance of the package is selected. The semantics of a function or procedure call are given by textually replacing the call by the body of the function or procedure itself, changing the names of formal parameters to their corresponding actual parameters, and renaming local variables with unique identifiers.

```
conditional_statement ::=
        if expression then statement_list
        { elsif expression then statement_list }
        [ else statement_list ]
        end if
```

One form of conditional statement is provided, having exactly the same form as the Ada **if** statement. However, to keep the specification language as simple as possible, no **case** statement is provided.

```
all_loop ::=
        all loop_variable in type_name loop
            statement_list
        end loop
for_loop ::=
        for loop_variable in type_name loop
            statement_list
        end loop
while_loop ::=
        while expression loop
            statement_list
        end loop
```

Three kinds of loop are provided: **all** loops, **for** loops, and **while** loops. The body of an **all** or **for** loop is executed with the loop variable assigned each value of the domain type. In a **for** loop, the values are taken in increasing order of the domain type. In an **all** loop, this restriction does not hold, and the instances of the loop body may be executed in any order, or in parallel.

Since the order of execution of a **for** loop is also a possible order of execution of an **all** loop, an **all** loop may always be safely replaced by a **for** loop. The reverse is not true. **For** loops are reserved for two purposes: in those rare situations when the order of execution matters, or when it is used specifically to describe the implementation of a sequential access process. An **all** loop is always preferred to a **for** loop whenever the instances of the loop body can be executed concurrently *in principle*, even if they are executed sequentially in practice.

There are actually two kinds of parallel loop, illustrated by the following two examples:

```
Stock : natural := 0;
...
all t in title loop
   Report.Put (C(t));
end loop;
all t in title loop
   Stock := Stock + C(t);
end loop;
```

In the first example, all the instances of the loop body can be executed at the same time. In the second, 'Stock' is a shared variable, which suggests that the instances ought to be executed one at a time. In fact, the value of 'Stock' can be computed by combining the values of 'C' pair-wise, e.g., for 8 elements, the computation is:

$$((C(1) + C(2))+(C(3) + C(4)))+((C(5) + C(6))+(C(7) + C(8)))$$

Four pairs of terms are summed, then two pairs of partial results, and finally one pair of partial results. The number of steps is $\log_2 N$, where N is the number of terms. This idea can be generalised to any associative operator, and is sometimes called 'reduce' or β-reduction [Hillis 1985]. The form given in the example, where an accumulator (e.g., 'Stock') is initialised in its declaration, then updated within an **all** loop, will be written to specify a reduction operation. Syntactically, a reduction has the form 'V := V op E;' or 'V := fn(V,E);', where 'V' is a variable, 'E' is an expression, 'op' is an operator, and 'fn' is a function. (The parallel implementation described in Chapter 3 actually finds a reduction by a different method.)

Given enough processors, an **all** loop with a shared variable can be executed in time $\mathbf{O}(\log N)$, but one without a shared variable can be executed in time $\mathbf{O}(1)$. In practice, the range of an **all** variable is likely to be so great that it will easily exceed the number of available processors. In that case, the speed-up due to parallelism is likely to be roughly equal to the number of processors, for either kind of loop.

Because Ada operators can be overloaded, it is not possible to tell when a **for** loop can be replaced by an **all** loop textually. In the second example above, there is no guarantee that '+' is an associative operator. In an extreme case, it could denote division, in which case a **for** loop would produce a unique final value for 'Stock', but an **all** loop could produce many alternative values. In addition, if array elements are distributed cyclically and partial reduction is done locally, as described later in Section 3.5, the operator must also be commutative. Therefore both keywords are needed.

```
with report;
generic
   type order is private;
   type employee is private;
   type product is private;
   type money is range <>;
package Sales is
   procedure Bonuses;
end Sales;
package body Sales is
   Agent : array (order) of employee := (others => 0);
   Item_Sold : array (order) of product := (others => null);
   Qty_Sold : array (order) of natural := (others => 0);
   Price : array (product) of money := (others => 0);
   Rate : array (product) of natural := (others => 0);
   procedure Bonuses is
      Bonus : array (employee) of money := 0;
      What : product := null;
      Value : money := 0;
      Rep : employee := null;
   begin
      all Ord in order loop
         What := Item_Sold(Ord);
         Value := Qty_Sold(Ord) * Price(What);
         Rep := Agent(Ord);
         Bonus(Rep) := Bonus(Rep) + Value * Rate(What)/1000;
      end loop;
      all Emp in employee loop
         Report.Bonuses(Emp, Bonus(Emp));
      end loop;
   end Bonuses;
end Sales;
```

EXAMPLE 2.6.1: SHARED VARIABLES

Because of the restriction that array indices must be simple variables, a difficulty can occasionally arise in specifying an **all** loop. Consider Example 2.6.1. The intention is that, given a set of orders, 'Bonuses' computes the commission payable to each sales representative. To do this, an array, 'Bonus', is created internal to the event procedure. Each entry, 'Bonus(Rep)', accumulates the commission for one sales representative. Several assignments to the same element may occur in parallel, but this is permissible, because accumulating a total is a reduction operation, and the assignments may be done in any order. However, in order to index the 'Bonus' array, it is first necessary to assign 'Agent(Ord)' to 'Rep', because the specification language does not allow a reference of the form 'Bonus(Agent(Ord))'. The problem is that one instance of 'Rep' is shared by all the instances of the loop body, and there is no guarantee that its value will survive from assignment to use. Creating multiple instances of 'Rep' by declaring it as an array does not resolve the problem.

To resolve it, the specification language allows variables to be declared within any statement, using a **declare** block. This is illustrated in Example 2.6.2. It is assumed that, because 'Rep' is declared within the first **all** loop, there is a separate instance of it for each instance of the loop body, therefore it is not shared.

```
package body Sales is
   Agent : array (order) of employee := (others => 0);
   Item_Sold : array (order) of product := (others => null);
   Qty_Sold : array (order) of natural := (others => 0);
   Price : array (product) of money := (others => 0);
   Rate : array (product) of natural := (others => 0);
   procedure Bonuses is
      Bonus : array (employee) of money := 0;
   begin
      all Ord in order loop
         declare
            What : product := null;
            Value : money := 0;
            Rep : employee := null;
         begin
            What := Item_Sold(Ord);
            Value := Qty_Sold(Ord) * Price(What);
            Rep := Agent(Ord);
            Bonus(Rep) := Bonus(Rep) + Value * Rate(What)/1000;
         end;
      end loop;
      all Emp in employee loop
         Report.Bonuses(Emp, Bonus(Emp));
      end loop;
   end Bonuses;
end Sales;
```

EXAMPLE 2.6.2: USING A DECLARE BLOCK

The **while** loop is provided for those cases where an iteration is needed that cannot be expressed using an **all** loop or **for** loop. The statement '**while** *expression* **loop** *statement* ; **end loop**;' is semantically equivalent to the infinite conditional:

```
if expression then
   statement ;
end if;
if expression then
   statement ;
end if;
...
```

and is a sequential construct whose instances cannot be evaluated concurrently.

2.7 Rewriting Rules

There are two rewriting rules, which enable the specification of a system to be reconstructed from the specifications of its component processes. Since these rules are straightforward, rather than describe them formally, they will be illustrated by example.

The first rule allows an array of packages to be reduced to a single package.

Suppose that the basis of the package array 'P' is '(T, U)', where 'T' and 'U' are type names, i.e., the package is declared using the syntax '**array** (T, U) **of package** P **is** ...'.

Then each simple state variable of 'P' should be converted to an array indexed by '(T, U)', and each array state variable should be given extra dimensions of 'T' and 'U'. For example, the declarations:

```
x : natural;
y : array (V) of string;
```

should be converted to:

```
x : array (T, U) of natural;
y : array (T, U, V) of string;
```

This example is easily generalised to any number of indices.

Second, each event procedure declaration and definition in the package should be given two additional parameters, of types 'T' and 'U', whose names do not clash with any others. For example, the event declaration:

```
procedure E (v : integer);
```

should be replaced by something like:

```
procedure E (t1 : T; u1 : U; v : integer);
```

Each reference to a state variable in the event definition should then be augmented by having these additional variables as indices. Again, this transformation generalises to any number of indices.

At each point where an event procedure is called from another package, the actual parameters specifying the member of the package array should be moved to follow the procedure name rather than the package name. For example, the call:

```
P (t, u) . E (v);
```

should be replaced by:

```
P . E (t, u, v).
```

Finally, the **array** clause should be removed from the package specification.

An example of this transformation was that from Example 2.5.3 to Example 2.5.2.

The second transformation enables a package to be deleted, after moving its contents to another package. There are three steps. First, calls to event procedures defined in the package to be removed must be replaced by their texts, after substituting actual for formal parameters. Second, the state variable declarations in the called package should be moved to the calling package. (If this should cause a clash with the name of a local variable, the local variable must be renamed.) Third, the called package specification and definition should be deleted.

Two such transformations will convert Example 2.5.2 into Example 2.5.1. The intermediate step is given in Example 2.7.1. This shows the effect of moving the contents of 'D' of

Example 2.5.2 into 'C'. The transformation of Example 2.7.1 into Example 2.5.1 is completed by moving the contents of 'C' into the enclosing 'Library' package.

The rewriting rules specify how to derive a system specification from its component specifications. They therefore provide a way of verifying that a given decomposition is correct. Conversely, their inverses allow many ways of decomposing a given system into components. Given an infra-structure that correctly replaces the system read loop, any decomposition that leads back to the original system specification is a correct one. Some suitable infra-structures will discussed in the next chapter. How to choose a good decomposition is a problem that will occupy a major part of the discussion that follows it.

```
package body Library is
   package C is
      procedure Buy (t : title);
      procedure Borrow (t : title; u : user);
      procedure Reshelve (t : title; u : user);
      procedure Audit;
   end C;
   package  body C is
      C : array (book) of natural := (others => 0);
      D : array (user) of natural := (others => 0);
      procedure Buy (t : title) is
      begin
         C(t) := C(t) + 1;
      end Buy;
      procedure Borrow (t : title; u : user) is
      begin
         if C(t) > 1 then
            D(u) := D(u) + 1;
            C(t) := C(t) – 1;
         end  if;
      end Borrow;
      procedure Reshelve (t : title; u : user) is
      begin
         D(u) := D(u) – 1;
         C(t) := C(t) + 1;
      end Reshelve;
      procedure  Audit is
         Loans, Stock: natural := 0;
      begin
         all t in title loop Stock := Stock + C(t); end loop;
         all u in user loop Loans := Loans + D(u); end loop;
         Report.Audit (S, L);
      end Audit;
   end C;
   procedure Buy (t : title) is
   begin
      C.Buy(v);
   end Buy;
   procedure Borrow (t : title; u : user) is
   begin
      C.Borrow(t, u);
   end Borrow;
   procedure Reshelve (t : title; u : user) is
   begin
      C.Reshelve(t, u);
   end Reshelve;
   procedure Audit (t : title; u : user) is
   begin
      C.Audit(t, u);
   end Audit;
end Library;
```

EXAMPLE 2.7.1: TRANSFORMATION OF LIBRARY COMPONENT SPECIFICATIONS

3. Implementation

The purpose of this chapter is to show, in specific terms, the technology of implementing a batch system. This is a digression from the main thrust of the thesis. Its purpose is to show that when a batch process uses sequential or parallel access, there are limitations on what it can do, which are exactly those expressed in Example 2.5.3a–b. Indeed, the limitations of sequential and parallel access are essentially the same, and the two approaches can be considered to be specific examples of the more general concept of independent access. The author believes that the inherent limitations of independent access are most easily understood and more convincingly presented by means of examples — otherwise the scheme of Example 2.5.3a–b might seem an arbitrary one. The sequential access algorithm given here is actually an extension of the usual one, to allow for global queries and updates, so it is important to show how this extension works. In the case of the parallel access system, it is also important to show that the parallelism results in an actual speed up, and that the advantages of parallel processing are not overwhelmed by the cost of communication between processes.

The three program examples given are each based on the Macrotopian Reference Library system of the preceding chapters. The first example uses random access, and is written in Cobol. The second uses sequential access, and is also written in Cobol. The third uses parallel access, and is written in *C* for the CM-5 parallel processor, using functions drawn from the CM-5's CMML message passing library [Thinking Machines 1993]. All three examples are paradigms that can be adapted to suit other systems; that is, they demonstrate alternative system read loops, and can be adapted as technology and circumstances permit.

Naturally, all three examples have common features. Event records have four fields. 'Timing' is the time or serial number of the event record, which reflects the temporal order of the real-world events. 'Kind' defines the type of the event: 'Borrow', 'Reshelve', 'Buy' or 'Audit'. 'User' specifies the identification of a library user, and 'Title' specifies the identification of a book. Where a field is not needed in a particular type of event, i.e., the 'User' in a 'Buy' event and both the 'User' and 'Title' in an 'Audit' event, it contains a value less than any actual name or title. (It would be better programming practice for each kind of event to have a different format, including only those fields it needs. A common format was adopted to keep the examples as simple as possible.)

In the Cobol examples, the 'Branches' file contains a record for each user who has drawn a positive number of books from the library. Each record has two fields: 'User' identifies the user, and 'Drawn' counts the number of books drawn by the user. The file is indexed by 'User'. The implementation uses the convention that a user who has drawn zero books does not need to be represented by a record. This is unrealistic in a more general context, because it does not keep track of the set of valid 'User' identifications. However, it is perfectly correct for the library system problem as specified. Likewise, the 'Books' file contains a record for each book that is in stock. Each record has two fields: 'Title' identifies the book, and 'Copies' counts the number of books in stock. The file is indexed by 'Title'. A book that is out of stock does not need to be represented by a record. This means that the system cannot keep track of the set of valid titles. In the parallel implementation, the representation is even simpler, 'Drawn' is an array indexed by 'User', and 'Copies' is an array indexed by 'Title', exactly as in the specification.

One of the more difficult aspects of the second and third examples is understanding how 'Audit' events are implemented. This is because the database is only in a consistent state before or after a batch of events is processed. The states of all the books are updated completely in response to other events before the states of any users are changed at all. Despite this, 'Audit' events are able to snapshot a consistent state of the inconsistent database at any desired time during the update. This is possible because they follow the same paths and are subject to the same delays as the updates that change its state.

3.1 Modelling Independent Updating

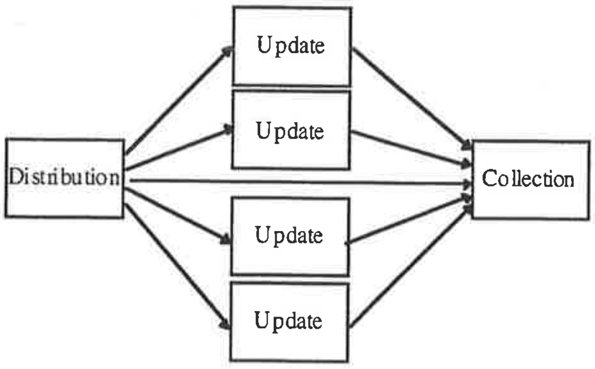

FIGURE 3.1.1: A MODEL OF INDEPENDENT ACCESS

Figure 3.1.1 shows the underlying structure of either a sequential or parallel update algorithm from the viewpoint of one event. (The Macrotopian Reference Library example needs two such updates.) There are many element update processes, one for each array element or key. In a parallel update these processes exist on separate physical processors — ideally, one to each processor; in a sequential update, they exist at different intervals of time. The element updates may inspect or update the database in whatever way is necessary. The distribution process sends messages to each element update, and the element updates send messages to a collection process. The element updates cannot communicate with one another; there is no guarantee that the messages they sent would arrive in time to be useful. In a parallel update, the destination process may have already processed updates with *later* time-stamps. In a sequential update, a message sent to an update process for a lower key value will arrive too late to be processed at all. This absence of communication between different element update processes is the defining property of the independent access model. It is precisely this restriction that allows each process to go about its work without interference or the need for synchronisation.

To relate this model to the specification of independent access given in Section 2.5, we note that its shell processes can be divided into three phases: an initial phase that occurs before the **all** loop, the independent access within the **all** loop, and a final phase that occurs after the **all** loop. Essentially, these phases correspond to the distribution, update and collection processes of Figure 3.1.1.

A 'distribution' process ensures that information concerning the event is sent to each element update process to which it is relevant. Two distribution patterns are illustrated by the library system. Updates are sent to one element update process, but 'Audits' are broadcast to all of them. Other possibilities exist. In principle, messages may be sent to any known list of element update processes. The difference between sequential and parallel processing is the means of distribution. In parallel processing, the means is to send messages from the processor supporting the distributor to the processors supporting the element updates. In sequential processing, the means is to tag a record with the key of an element, and to sort the records. When the event records are merged sequentially with the master file, the record will appear in main memory at exactly the right time. In the case of a broadcast, a 'memo' is created that stays in main memory throughout the update. In principle, such memos could be used for all events, but they would use up too much main memory. Sorting events allows them to be held in secondary storage and read into main memory as needed, with low cost. If there happens to be a known list of elements that must be inspected or updated, records can be created for each element in the list, and sorted. These records correspond to procedure calls made by a 'shell' process to the elementary update processes it contains — although a common case is that the shell merely passes a call made to it to the single element of interest.

The 'collection' process accepts messages sent by the element update processes. There may be one such message, as in the case of the library system updates, or one from each process, as in an 'Audit' event. Intermediate situations are possible. Messages could arrive from any list of element update processes selected by the distribution process, and be processed in any way desired. Sometimes the collection process is degenerate, and does nothing. The sequential access version of the library system has no obvious collection processes in it. In the case of 'Audits', this is because the 'Stock' and 'Loans' variables can be safely updated by the element update processes. In the case of updates, all information concerning the update is contained in a single event record. In general however, it would be possible to sort messages received from several element update processes by time, thus collecting all the records for a given event.

Figure 3.1.1 shows the viewpoint of one event. Logically, each event has its own distributor and collector, although they all share the same set of element update processes. In the parallel implementation of the library system, these distributors and collectors are distributed evenly over the available physical processors. In the sequential implementation, they are scarcely evident, because distributing and collecting results is trivial.

To achieve greater generality in a sequential update, a distributor process would need to be invoked to pre-process the events before the sort operation. This would provide an opportunity to distribute events to as many update processes as desired, by creating several records to be matched against different master records. Conversely, when all the element updates had been completed, the transfer file they produced would be sorted by time, so that all records relevant to an event would be collected together. The collector processes for each event would then post-process these collected records. Such a collecting sort is not a common part of a sequential update algorithm in practice. As in the library example, collection processes are often degenerate. The sort may then be dropped; there is no reason for one update phase to sort its out-going transfer records into time order when the next update will begin by sorting them into key order. Nonetheless, both pre-processing and post-processing may be needed in general.

Since there is exactly one distributor and one collector for an event, it is possible to think of the collector as an extension of the distributor, so that together they form what in Example 2.5.3a–b was called a 'shell'. The messages sent by the distributor to the element updates can be represented as procedure calls. The messages sent by element update processes to the collector can be treated as values returned by these calls. This justifies the abstraction given in Example 2.5.3a–b of modelling communication between the shell and the element updates as ordinary procedure calls.

The model of Figure 3.1.1 restricts the possible forms of a shell procedure. A shell procedure can consist of some pre-processing (in the distributor), a series of calls to element updates (distribution), followed by post-processing (in the collector). There is just one opportunity to do each of these things. It would not be permitted, for example, to call an element update and use the returned result as a parameter of a subsequent call to another element update. It would create a dependence between the two element updates that would destroy their independence.

3.2 Random Access Implementation

The first example is the most straightforward. Example 3.2.1a shows the first three divisions of a Cobol program to implement the library system. Example 3.2.1b shows the main logic of the procedure division of the library system, and Example 3.2.1c shows the subsidiary procedures needed to support input and output operations.

In Example 3.2.1b, 'System-Read-Loop' implements the generic event loop of Example 2.1.2. The remainder of Example 3.2.1b consists of the four event procedures. They follow the specification closely, except that they replace its array references with file input and output operations.

```
Identification Division.
Program-ID.                 Library-System.
Environment Division.
Input-Output Section.
File-Control.
    Select Events assign "events"; organization is sequential.
    Select Books assign "books"; organization is indexed,
        record key is Title of Books; access is dynamic.
    Select Branches assign "branches"; organization is indexed,
        record key is User of Branches; access is dynamic.
Data Division.
FD Events.
    1   Event.
        88  At-Start            value low-values.
        88  At-End              value high-values.
        2   Timing              picture 9(4).
        2   Kind                picture 9.
            88  Buy             value 1.
            88  Borrow          value 2.
            88  Reshelve        value 3.
            88  Audit           value 4.
        2   User                picture x(10).
        2   Title               picture x(20).
FD Books.
    1   Book.
        88  At-Start            value low-values.
        88  At-End              value high-values.
        2   Title               picture x(20).
        2   Copies              picture 9(2).
FD Branches.
    1   Branch.
        88  At-Start            value low-values.
        88  At-End              value high-values.
        2   User                picture x(10).
        2   Drawn               picture 9(2).
Working-Storage Section.
    77  Stock                   picture 9(4) value zero.
    77  Loans                   picture 9(4) value zero.
    77  Book-Status             picture 9.
        88  Book-Missing        value 0.
        88  Book-Exists         value 1.
    77  Branch-Status           picture 9.
        88  Branch-Missing      value 0.
        88  Branch-Exists       value 1.
```

EXAMPLE 3.2.1A: RANDOM ACCESS LIBRARY SYSTEM IN COBOL (PART 1)

```
Procedure Division.
System-Read-Loop.
    Open input Events, I-O Books, Branches;
    Read Events, at end, Set At-End of Events to true; end-read;
    Perform until At-End of Events,
        Evaluate Kind of Event;
            when Buy, Perform Process-Buy;
            when Borrow, Perform Process-Borrow;
            when Reshelve, Perform Process-Reshelve;
            when Audit, Perform Process-Audit;
        end-evaluate;
        Read Events, at end Set At-End of Events to true; end-read;
    end-perform;
    Close Events, Books, Branches;
    Stop run.
Process-Buy.
    Perform Read-Book;
    Add 1 to Copies of Book;
    Perform Update-Book.
Process-Borrow.
    Perform Read-Book;
    If Copies of Book > 1 then
        Perform Read-Branch;
        Add 1 to Drawn of Branch;
        Perform Update-Branch;
        Subtract 1 from Copies of Book;
        Perform Update-Book;
    end-if.
Process-Reshelve.
    Perform Read-Branch;
    Subtract 1 from Drawn of Branch;
    Perform Update-Branch;
    Perform Read-Book;
    Add 1 to Copies of Book;
    Perform Update-Book.
Process-Audit.
    Perform Read-First-Book;
    Perform until At-End of Books,
        Add Copies of Book to Stock;
        Read Books, at end Set At-End of Books to true; end-read;
    end-perform;
    Perform Read-First-Branch;
    Perform until At-End of Branches,
        Add Drawn of Branch to Loans;
        Read Branches, at end Set At-End of Branches to true; end-read;
    end-perform;
    Display Stock, " books in stock, ", Loans, " books on loan.".
```

EXAMPLE 3.2.1B: RANDOM ACCESS LIBRARY SYSTEM IN COBOL (PART 2)

```
Read-Book.
    Move Title of Event to Title of Book;
    Read Books,
    invalid key Move zero to Copies of Book; Set Book-Missing to true;
    not invalid, Set Book-Exists to true;
    end-read.
Update-Book.
    If Book-Exists then
        If Copies of Book = zero then
            Delete Books, invalid key Perform I-O-Error; end-delete;
        else
            Rewrite Book, invalid key Perform I-O-Error; end-rewrite;
        end-if;
    else
        If Copies of Book not = zero then
            Write Book, invalid key Perform I-O-Error; end-write;
        end-if;
    end-if.
Read-Branch.
    Move User of Event to User of Branch;
    Read Branches,
    invalid key Move zero to Drawn of Branch; Set Branch-Missing to true;
    not invalid, Set Branch-Exists to true;
    end-read.
Update-Branch.
    If Branch-Exists then
        If Drawn of Branch = zero then
            Delete Branches, invalid key, Perform I-O-Error; end-delete;
        else
            Rewrite Branch, invalid key, Perform I-O-Error; end-rewrite;
        end-if;
    else
        If Drawn of Branch not = zero then
            Write Branch, invalid key, Perform I-O-Error; end-write;
        end-if;
    end-if.
Read-First-Book.
    Set At-Start of Books to true;
    Start Books, key not < Title of Books,
    invalid key, Set At-End of Books to true;
    not invalid, Read Books, at end Set At-End of Books to true; end-read;
    end-start.
Read-First-Branch.
    Set At-Start of Books to true;
    Start Branches, key not < User of Branches,
    invalid key, Set At-End of Branches to true;
    not invalid,
        Read Branches, at end Set At-End of Branches to true; end-read;
    end-start.
I-O-Error.
    Display "Invalid Key error detected when it is logically impossible!";
    Stop run.
```

EXAMPLE 3.2.1C: RANDOM ACCESS LIBRARY SYSTEM IN COBOL (PART 3)

Among other things, the procedures of Example 3.2.1c enforce the conventions surrounding default values. When an attempt is made to read a non-existent record, a default-valued record is supplied instead. Similarly, if an existing record is given a default value, it is deleted from the file. This matches the implementation of functional dependencies discussed in Section 2.4.

The first point to observe about the example is the close correspondence between its event procedures and those of the specification of Example 2.5.1. The second point is that this simple, direct form of implementation puts no restriction on the set of specifications that can be implemented; the implementation has the same form as the specification. It is a fall-back method that can always be used when other methods fail.

3.3 Sequential Access Implementation

The second implementation is more complex, and is based on the multi-process decomposition of Example 2.5.3. The system is split into 'C' and 'D' processes, called here 'Library-Books-Process' and 'Library-Branches-Process', which are written as separate programs. These two programs can be run at different times, with the intermediate results (i.e., the delayed procedure calls) being stored in a transfer file. Because the transfer file is written in 'extend' mode, it is possible to run the books process several times before running the branches process. For example, the books process could be run once a day and the branches process could be run once a week. This would serve no useful purpose in a library, but it is a common practice in situations where different reports are needed with different frequencies.

Example 3.3.1a shows the first three divisions of the implementation of the books process in Cobol.

The program's working storage contains an array, 'Memo', which is used to store the times at which 'Audit' events occur, and to accumulate the numbers of books in stock. The size of this array limits the number of 'Audit' events that can be present in one batch. This array is not a usual feature of an update program. Most programs in practice would make provision for at most one 'Audit' event. The reason for this is that batch programs are typically executed as soon as an important query is made. Indeed, the need for timely information often determines the size of a batch, so that, for example, if a report is required daily, a batch is processed once a day. The point shown by the example is that one query is not a limit in principle, and any bounded number of similar 'global' queries or updates can be processed in one batch. If a batch happened to contain more 'Audit' events than could be stored in the 'Memo' array, it would simply be split into two or more smaller batches.

```
Identification Division.
Program-ID.                 Library-Books-Process.
Environment Division.
Input-Output Section.
File-Control.
    Select Events-In        assign "events"; organization is sequential.
    Select Work-File        assign "temp1".
    Select Events           assign "temp2"; organization is sequential.
    Select optional Transfers  assign "transfer"; organization is sequential.
    Select Old-Books        assign "oldbooks"; organization is sequential.
    Select New-Books        assign "newbooks"; organization is sequential.
Data Division.
FD  Events-In.
    1   Event-In.
        2   Timing              picture 9(4).
        2   Kind                picture 9.
        2   User                picture x(10).
        2   Title               picture x(20).
SD  Work-File.
    1   Work-Record.
        2   Timing              picture 9(4).
        2   Kind                picture 9.
        2   User                picture x(10).
        2   Title               picture x(20).
FD  Events.
    1   Event.
            88  At-End          value high-values.
        2   Timing              picture 9(4).
        2   Kind                picture 9.
            88  Buy             value 1.
            88  Borrow          value 2.
            88  Reshelve        value 3.
            88  Audit           value 4.
        2   User                picture x(10).
        2   Title               picture x(20).
FD  Transfers.
    1   Transfer.
        2   Timing              picture 9(4).
        2   Kind                picture 9.
        2   User                picture x(10).
        2   Stock               picture 9(4).
FD  Old-Books.
    1   Old-Book.
        2   Title               picture x(20).
            88  At-End          value high-values.
        2   Copies              picture 9(2).
FD  New-Books.
    1   New-Book.
        2   Title               picture x(20).
            88  At-End          value high-values.
        2   Copies              picture 9(2).
Working-Storage Section.
    1   Memo                occurs 1000 times depending on Memo-Counter.
        2   Timing              picture 9(4) value zero.
        2   Stock               picture 9(4) value zero.
    77  Memo-Counter            picture 9(4) value zero.
    77  Current-Memo            picture 9(4) value zero.
```

EXAMPLE 3.3.1A: SEQUENTIAL ACCESS LIBRARY SYSTEM IN COBOL (PART 1)

```
Procedure Division.
Update-Books.
    Sort Work-File,
        on ascending key Title of Work-Record, Timing of Work-Record,
        using Events-In, giving Events;
    Open input Events, Old-Books, output New-Books, extend Transfers;
    Read Events, at end Set At-End of Events to true; end-read;
    Move zero to Memo-Counter;
    Perform until Title of Event not = low-values,
        Add 1 to Memo-Counter;
        Move Timing of Event to Timing of Memo (Memo-Counter);
        Read Events, at end Set At-End of Events to true; end-read;
    end-perform;
    Read Old-Books, at end Set At-End of Old-Books to true; end-read;
    Perform Choose-New-Title;
    Perform Process-One-Title until At-End of New-Books
    Perform Process-Audit-2, varying Current-Memo from 1,
        by 1 until Current-Memo > Memo-Counter;
    Close Events, Old-Books, New-Books, Transfers;
    Stop run.
Process-One-Title.
    If Title of Old-Book = Title of New-Book then
        Move Copies of Old-Book to Copies of New-Book;
        Read Old-Books, at end Set At-End of Old-Books to true; end-read;
    else
        Move zero to Copies of New-Book;
    end-if;
    Move 1 to Current-Memo;
    Perform Process-One-Event until Title of Event not = Title of New-Book,
    Perform Process-Audit-1 varying Current-Memo from Current-Memo by 1,
            until Current-Memo > Memo-Counter;
    If Copies of New-Book not = zero then Write New-Book; end-if;
    Perform Choose-New-Title.
Process-One-Event.
    Perform until Current-Memo > Memo-Counter
            or Timing of Memo (Current-Memo) not < Timing of Event,
        Perform Process-Audit-1;
        Add 1 to Current-Memo;
    end-perform;
    Evaluate Kind of Event;
        when Buy, Perform Process-Buy;
        when Borrow, Perform Process-Borrow;
        when Reshelve, Perform Process-Reshelve;
    end-evaluate;
    Read Events, at end Set At-End of Events to true; end-read.
Choose-New-Title.
    If Title of Event < Title of Old-Book then
        Move Title of Event to Title of New-Book;
    else
        Move Title of Old-Book to Title of New-Book;
    end-if;
```

EXAMPLE 3.3.1B: SEQUENTIAL ACCESS LIBRARY SYSTEM IN COBOL (PART 2)

Example 3.3.1b shows the input-output logic of the procedure division. It is based on the well-known algorithm in [Dijkstra 1976 , Dwyer 1981a, Inglis 1981, Dwyer 1981b], extended to deal with global queries. It is a generic paradigm that can be adapted for any sequential update.

In the procedure 'Update-Books', the input events are sorted into title order, and where two events concern the same title, their original time ordering is preserved. After the sort, the 'Audit' events (which are given dummy low-valued titles) are collected to the beginning of the sorted file. The program then reads their associated times into the 'Memo' array.

The procedure then enters a loop that is iterated for each value of 'title' that appears on either the old books file or in the event transfer file, calling the 'Process-One-Title' procedure.

The underlying logic of 'Process-One-Title' is to find the initial state of the title from the 'Old-Books' file, update or inspect its state according to the events file, then record its final state on the 'New-Books' file. First, the existing number of copies of the title is found from the old books file — but if there is no record for the current title, it is assumed to have zero copies. Second, the procedure enters a loop where the state of the book (its number of copies) is updated by 'Process-One-Event'. Third and last, the number of copies is recorded in the new book file — but only if it is non-zero. The final state is then inspected by any 'Audit' events that follow the last update, the earlier inspections having being dealt with inside 'Process-One-Event'.

Each execution of 'Process-One-Event' corresponds to a 'Buy', 'Borrow' or 'Reshelve' event, so in this example, each execution always corresponds to an update event causing a state transition. However, because of the 'Audit' events, there may be any number of inspections needed between updates. Therefore, before each update event, the 'Audit' events that immediately precede it are processed by stepping through the 'Memo' array.

The overall processing of an 'Audit' event is as follows. The value of 'Stock' in the 'Memo' array is initialised to zero at the start of the program. This matches the similar initialisation in process 'C' of the specification of Example 2.5.3. The procedure 'Audit-1' matches the assignment statement in the **all** loop of 'C'. The procedure 'Audit-2' matches the procedure call of 'D.Audit'.

```
Process-Buy.
    Add 1 to Copies of New-Book.
Process-Borrow.
    If Copies of New-Book > 1 then
        Write Transfer;
        Subtract 1 from Copies of New-Book;
        Move corresponding Event to Transfer;
    end-if.
Process-Reshelve.
    Write Transfer;
    Add 1 to Copies of New-Book;
    Move corresponding Event to Transfer.
Process-Audit-1.
    Add Copies of New-Book to Stock of Memo (Current-Memo).
Process-Audit-2.
    Set Audit of Transfer to true;
    Move corresponding Memo (Current-Memo) to Transfer;
    Move low-values to User of Transfer;
    Write Transfer.
```

EXAMPLE 3.3.1C: SEQUENTIAL ACCESS LIBRARY SYSTEM IN COBOL (PART 3)

Finally, Example 3.3.1c shows the application specific statements corresponding to the event procedures in the specification of Example 2.5.3. Where the specification calls for delayed procedure calls, the procedures write a transfer record with suitable contents. The example is degenerate in that no event procedure makes more than one delayed call, although the specification rewriting rules may demand several calls in general. When several calls do occur, it is important for their order to be preserved when they are sorted by the downstream process. Since two calls generated by the same event share the same value of 'Timing', in general the 'Timing' field of transfer records should be augmented by an additional 'call serial number' to preserve their correct sequence, making something similar to a Dewey number.

There are two kinds of event procedure that can be processed in the sequential update, those that concern one particular title, and global events that concern all titles — signalled in the specification language by an **all** loop. Because those that involve all titles must be present throughout the update, they must be held in a buffer ('Memo'), so their number is limited by the size of the buffer — often 1 in practice. On the other hand, those that concern single titles require no special buffer, and their number is limited only by the size of the file system. The example is misleading however, in that only one kind of global event is present. In general, the loops in the example that concern the 'Audit' events would need to allow for more than one kind of event. Also, all the single title events are updates, and the only global event is a query. This need not be the case. With a different example, a banking system say, it is easy to imagine a global update, to add a month's interest to each account, for example. Conversely, a query event could report the balance of a single account.

```
Identification Division.
Program-ID.                 Library-Branches-Process.
Environment Division.
Input-Output Section.
File-Control.
    Select Transfers      assign "transfers"; organization is sequential.
    Select Work-File      assign "temp1".
    Select Events         assign "temp2"; organization is sequential.
    Select Old-Branches   assign "oldbranches"; organization is sequential.
    Select New-Branches   assign "newbranches"; organization is sequential.
Data Division.
FD Transfers.
    1  Transfer.
        2   Timing              picture 9(4).
        2   Kind                picture 9.
        2   User                picture x(10).
        2   Stock               picture 9(4).
SD Work-File.
    1  Work-Record.
        2   Timing              picture 9(4).
        2   Kind                picture 9.
        2   User                picture x(10).
        2   Stock               picture 9(4).
FD Events.
    1  Event.
            88  At-End          value high-values.
        2   Timing              picture 9(4).
        2   Kind                picture 9.
            88  Buy             value 1.
            88  Borrow          value 2.
            88  Reshelve        value 3.
            88  Audit           value 4.
        2   User                picture x(10).
        2   Stock               picture 9(4).
FD Old-Branches.
    1  Branch.
        2   User                picture x(10).
            88  At-End          value high-values.
        2   Drawn               picture 9(2).
FD New-Branches.
    1  Branch.
        2   User                picture x(10).
            88  At-End          value high-values.
        2   Drawn               picture 9(2).
Working-Storage Section.
    1  Memo               occurs 1000 times depending on Memo-Counter.
        2   Timing              picture 9(4) value zero.
        2   Stock               picture 9(4) value zero.
        2   Loans               picture 9(4) value zero.
    77 Memo-Counter             picture 9(4) value zero.
    77 Current-Memo             picture 9(4) value zero.
```

EXAMPLE 3.3.2: SEQUENTIAL ACCESS LIBRARY SYSTEM IN COBOL (PART 4)

```
 Procedure  Division.
 Update-Branches.
     Sort Work-File,
         on ascending key User of Work-Record, Timing of Work-Record,
         using Transfers, giving Events;
     Open input Events, Old-Branches, output New-Branches;
     Read Events, at end Set At-End of Events to true; end-read;
     Move zero to Memo-Counter;
     Perform until User of Event not = low-values,
         Add 1 to Memo-Counter;
         Move corresponding Event to Memo (Memo-Counter);
         Read Events, at end Set At-End of Events to true; end-read;
     end-perform;
     Read Old-Branches, at end Set At-End of Old-Branches to true; end-read;
     Perform Choose-New-User;
     Perform Process-One-User until At-End of New-Branches;
     Perform Process-Audit-3, varying Current-Memo from 1,
         by 1 until Current-Memo > Memo-Counter;
     Close Events, Old-Branches, New-Branches;
     Stop  run.
 Process-One-User.
     If User of Old-Branch = User of New-Branch then
         Move Drawn of Old-Branch to Drawn of New-Branch;
         Read Old-Branches, at end
               Set At-End of Old-Branches to true;
         end-read;
     else
         Move zero to Drawn of New-Branch;
     end-if;
     Move 1 to Current-Memo;
     Perform Process-One-Event until User of Event not = User of New-Branch;
     Perform Process-Audit-1 varying Current-Memo from Current-Memo by 1,
         until Current-Memo > Memo-Counter;
     If Drawn of New-Branch not = zero then Write New-Branch; end-if;
     Perform Choose-New-User.
 Process-One-Event.
     Perform until Current-Memo > Memo-Counter
               or Timing of Memo (Current-Memo) > Timing of Event,
         Perform Process-Audit-1;
         Add 1 to Current-Memo;
     end-perform;
     Evaluate Kind of Event;
         when Borrow, Perform Process-Borrow;
         when Reshelve, Perform Process-Reshelve;
     end-evaluate;
     Read Events, at end Set At-End of Events to true; end-read.
 Choose-New-User.
     If User of Event < User of Old-Branch then
         Move User of Event to User of New-Branch;
     else
         Move User of Old-Branch to User of New-Branch;
     end-if;
```

EXAMPLE 3.3.2B: SEQUENTIAL ACCESS LIBRARY SYSTEM IN COBOL (PART 5)

```
Process-Borrow.
     Add 1 to Drawn of New-Branch.
Process-Reshelve.
     Subtract 1 from Drawn of New-Branch.
Process-Audit-1.
     Add Drawn of New-Branch to Loans of Memo (Current-Memo).
Process-Audit-2.
     Display Stock of Memo (Current-Memo), " books in stock, ",
               Loans of Memo (Current-Memo), " books on loan.".
```

EXAMPLE 3.3.2C: SEQUENTIAL ACCESS LIBRARY SYSTEM IN COBOL (PART 6)

Example 3.3.2a shows the first three divisions of the program corresponding to process 'D' of Example 2.5.3. They are directly analogous to first three divisions of 'Update-Books' except that, because it is the last step in the pipeline, 'Update-Branches' has no output transfer file. (The analogy and the correspondence with the specification would be improved if the 'display' statements that report the results of the 'Audit' queries were replaced by write statements to a transfer file, to be read by a 'Report' package.)

Example 3.3.2b shows the sequential file update logic. It is strikingly similar to Example 3.3.1b. It is clear that it could be generated automatically from the specification.

Finally, Example 3.3.2c shows the application specific procedures, corresponding to the event procedures of process 'D' in Example 2.5.3. Apart from the substitution of the 'display' statement for a delayed procedure call to an external system, there is nothing new to describe.

3.4 Analysis of Sequential and Random Access

Why is sequential access preferred to random access? — or rather, *when* is it preferred?

Sequential access can out-perform random access simply because the logical records of a file or the rows of a database are physically stored in blocks or pages of many records. This follows from the nature of rotating storage media. The access time for a magnetic disk drive is made of three components: the time taken to move the read-write head to the correct track (seek time), the time taken for the desired block to rotate to the read-write head (latency), and the time taken for the block to pass under the head (transfer time). At the time of writing, average seek times are of the order of 10mS and average latency is about 5mS. Transfer time depends on the size of a block, and for the longest block that can fit on a track (say 50KB), it is double the average latency. Typical transfer rates are at least 1MB/sec. The point is, if it will take 15mS to start reading a record, it is foolish to read only a few bytes; it is better to spend 25mS to read 50K bytes. (A secondary reason is that the gaps between sectors on a disk waste space, so sectors are rarely less than 512 bytes long. It is impossible to read less than one sector.)

When a file is read sequentially, all the records in a block will be processed one after another. However, when a record is read at random, a whole block must be read, and the other records in it are rarely useful. Random access demands that blocks should be small, i.e., one sector.

Sequential access demands that they should be as long as possible. Given sufficient buffer space, they should occupy a whole track, or even a whole cylinder. (The upper limit may be the unit the operating system uses to allocate file space.) As a simple example, suppose a 1,000,000 byte file consists of 10,000 records of 100 bytes. Reading the file in one continuous transfer would take less than a second. At the other extreme, reading each record separately would take about 150 seconds.

It is easy to estimate when sequential access is faster than random access. When a batch contains global events like 'Audit' there is really no contest, as just one such event needs the whole master file to be read anyway. Assume instead that all events update single records, as 'Buy', 'Borrow' and 'Reshelve' do. Consider the number of master file accesses. In a sequential update, each master block must be read and written exactly once, regardless of the number of events. The number of accesses is therefore twice the number of master blocks. On the other hand, using random access, each update event causes one block of the master file to be read and rewritten, causing 2 disk accesses. The number of accesses is therefore twice the number of updates. If U is the number of updates and B is the number of master file blocks, the break-even point is given by

$$2U = 2B.$$

(This analysis is approximate. Occasionally a desired record will already be in the buffer, which saves an access; but on other occasions records must be inserted or deleted, which causes extra house-keeping accesses. We assume too that the file's index and memory map are present in main memory, making it possible to read or write a record in one shot. We also ignore the necessary precaution of taking a back-up copy of the file.)

The break-even point between the two methods is therefore where the number of updates equals the number of master blocks [Smith & Barnes 1987]. For fewer updates random access is faster; for more, sequential access is faster. This rule of thumb can be put into a form that stresses the importance of the blocking factor. Dividing both sides of the break-even equation by the number of master file records, M, the ratio of the number of updates to the number of master records (called the 'hit rate') should equal the inverse of the blocking factor:

$$U/M = B/M = (M/B)^{-1}.$$

For example, if the blocking factor is 20 records per block, sequential access is better provided the hit rate exceeds 5%. If the blocking factor is 100, the hit rate needs only to exceed 1%. This result is a surprise to the uninitiated, because intuitively it seems that it is inefficient to copy an entire file when only a small fraction of its records are updated.

It remains to justify having ignored reading and sorting the event transfer file. First, given the low hit rates involved at a typical break-even point, the event file is relatively small and has only a small effect. Second, it is read sequentially in both cases; the only difference is the cost of sorting. Here, two factors minimise its importance. First, the number of events at the break-even point is usually so small that the event file can be sorted in main memory without the use of a work file. Second, even if the sort cannot be carried out in main memory, modern sorting algorithms are very efficient, and operate by sequentially merging files. Even a very large batch could be sorted in one or two merge steps. Since at the break-even point the size of the event file is smaller than the master file by a ratio equal to the blocking factor, the sum of these steps can usually be ignored too. However, since the number of merge steps does grow logarithmically with the size of the batch, it is possible that for an extremely large batch that the cost of sorting could tip the balance in favour of random access. The cure is simple: split the batch into several smaller batches of optimum size.

These considerations are summarised in Figure 3.4.1. The horizontal axis plots the number of events in a batch. The vertical axis plots the number of disk accesses needed to process the batch. The graph for random access is a straight line passing through the origin because each event incurs an equal cost. The graph for sequential access has a vertical intercept that represents the fixed cost of copying the master file. It then ascends with a gentle slope, which is the cost of sorting and reading the event file. Its slope is much less than the slope for random access, because the event file is always read sequentially, and each access reads many events. The discontinuity in the graph represents a batch size where the sorting algorithm cannot operate within main memory, and has to merge external work files. After each discontinuity, the slope of the line increases, and would eventually overtake the random access line. Practical batch sizes fall well below this point.

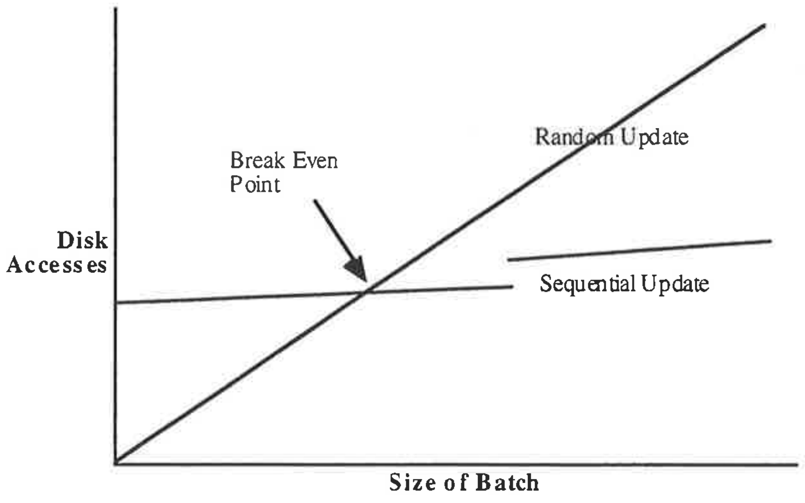

FIGURE 3.4.1: THE BREAK EVEN BETWEEN RANDOM AND SEQUENTIAL UPDATING

There is a temptation to combine sequential and random access updates in some way to gain the efficiency of sequential access with the flexibility of random access. This turns out to be possible only in limited circumstances.

Suppose we attempt to combine sequential access to the 'Branches' file with random access to the 'Books' file. This means that the events must be sorted into 'User' sequence. If there are several events that update a particular book, they will be processed in 'User' order rather than time order. This means that if two branch libraries try to borrow the last copy of a book, it is the branch with the lesser 'User' identifier that will get it, rather than the branch who requested it first. The system would not preserve its equivalence to the specification or to its direct implementation in Example 3.2.1. We say that it is not 'real time equivalent'.

In this example, processing the book file sequentially and the branches file randomly works a little better, because the only operations on the branches file are to increment and decrement 'Drawn'. The final state of 'Drawn' does not depend on the order in which the operations occur. However, its history may be different, and it may altogether fail to pass through some intermediate states that should be inspected by 'Audit' events. Indeed, with mixed random and sequential access it is impossible to devise any valid way to implement 'Audit' events.

There are two situations in which a mixed-mode update works correctly. The first is when the random access file is read-only, so that its state is constant. But when this is true, it is equally possible to look up the constant data in a separate sequential access that precedes the update in question, and this would usually be a more efficient implementation. An exception that has some value in practice occurs when the look-up file is small enough to be read into main storage as an array. It may then be accessed very quickly. By eliminating the need for a sort operation, this is preferable to a separate look-up step.

The second situation occurs when a file has a structured or composite key. This possibility will be discussed in Section 3.7.

3.5 A Parallel Update Algorithm

This section describes how the decomposition of the library system can be implemented as a parallel computer program using a CM-5 parallel processor. To understand the example, it is first necessary to understand the architecture of the CM-5.

The CM-5 contains a number of identical parallel processors — 32 in the case of the computer actually used. A separate front-end 'host' processor loads the parallel processors with identical copies of the same program. This means they have identical memory maps, which simplifies inter-process communication. The processors need not execute the same instructions however, because each processor has its own identification number. Each processor can learn its own identity and branch to a section of code specific to itself.

The processors of a CM-5 are interconnected by two message switching networks: the data network, and the control network. The data network is used for large message packets, and the control network is used for short messages, mainly to control synchronisation. The control network has some built-in logical capability. Both networks are organised as trees. Among other things, the control network can find a global sum using the tree reduction algorithm described in Section 2.6.4. The tree structures are invisible to the application program, which may send messages from point to point between any pair of processors. Messages are transmitted up the tree to the common ancestor of the sending and receiving processors. If messages involve separate sub-trees, they can be transmitted concurrently without interference.

However, in a random pattern of communication, one-half of all messages must pass through the root. Accordingly, the band-width of the tree increases from the leaves towards the root.

The basic plan was to distribute the 'C' and 'D' arrays cyclically across the 32 processors. This should allow parallel access to their elements, potentially speeding up access to the arrays by a factor of 32.

A problem with any parallel algorithm is to obtain sufficient *granularity*: the size of the task given to each processor must be big enough to justify the overhead of inter-processor communication. This proved to be a major issue. The CM-5 has a 50MHz clock (20nS per cycle), and the latency for the lowest-level of message passing proved to be about 30μS, a ratio of over 1,000:1. Because all the events except 'Audit' execute only a few instructions, their execution could easily be swamped by the message passing overhead.

A second problem is the initial distribution of the events to the processors. If all the events in a batch were initially stored on one processor, the act of distributing them could cause a serious bottleneck at the sending processor. In the implementation that follows, it is assumed that the events are already distributed evenly. In practice, this would be a reasonable assumption provided each processor served a similar number of client terminals.

The main technical challenge in programming the library system was to avoid the bottleneck of having a single process accumulate the results of summation, as in the case of the decomposition of Example 2.5.3a–b. One possibility would be to have the CM-5's control network compute the sums, but unfortunately this option was not compatible with other aspects of the implementation. No compiler was available that would support parallel summation alongside low-level message passing. Further, summation via the control network would require all the processors to become synchronised before each sum could be calculated. This would block other events, effectively dividing the input stream into a series of short batches separated by 'Audit' events. Instead, partial sums were accumulated on each processor independently, then forwarded to a collecting processor. A different collecting processor was assigned cyclically to each 'Audit' event in turn. In this way, each processor was given an equal load, which is the best that can be done.

The execution of the algorithm is divided into six well-defined phases corresponding to the distribution, update, and collection phases of the '{C}' and '{D}' processes. Figure 3.5.1 shows how the implementation is organised. Each processor assumes the roles of 'C distributor', 'C update', 'C collector', 'D distributor', 'D update' and 'D collector' in turn. (Figure 3.5.1 shows 3 rather than 32 processors.) Messages are passed between any two successive processes in the pipe-line, except between a 'C collector' and a 'D distributor'. This is because each 'C collector' assumes the role of 'D distributor' without the need for message passing.

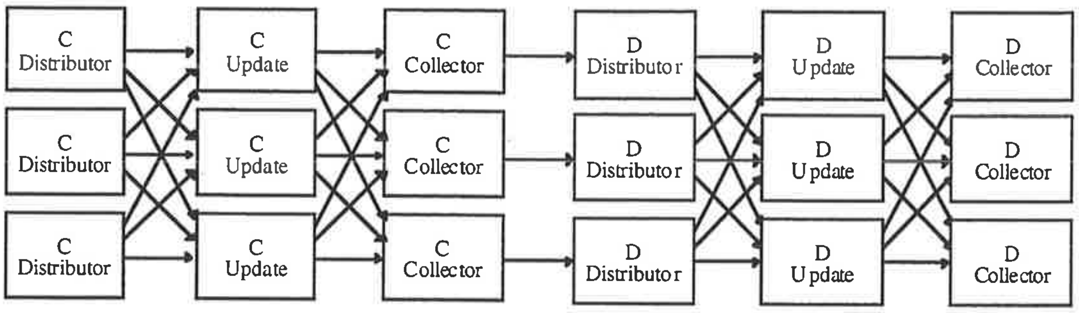

FIGURE 3.5.1: A MODEL OF THE IMPLEMENTATION

For ease of programming, no processor starts a new phase until every processor has completed its current one. This ensures that all outstanding messages have been received before each phase is wound up. The phase boundaries are enforced by a 'synchronise' library function that uses the control network.

Once the event streams are set up, each processor functions as a 'C distributor' by executing its 'C distribution' phase, sending the events it is allocated to the 'C update' process concerned. For all except the 'Audit' events, the update processor is computed from the value of the 'title', modulo 32. The 'Audit' events have to be broadcast to all the 'C update' processes.

The second, 'C update' phase, consists of two steps: first the events received by each 'C update' process are sorted into time sequence, then used to update or inspect $^1/_{32}$ of the 'C' array. Sorting is needed because inputs can arrive from 32 sources. Otherwise it would be possible for one 'Borrow' event to overtake an earlier one, and the wrong branch could then withdraw the last copy of a book. In processing an 'Audit' event, each processor can only form a partial sum, since it has access to only one partition of the array.

The third phase, which begins only when all the 'C updates' have completed, is called 'C collection'. In this phase each update processor sends the partial sums it has sampled for 'Audit' events to their 'C collector' processes. Each collector then adds the sums it receives to its own sum, computing the stock of books at the time of the audit. No action is needed for other events, because their 'C collector' processes are degenerate.

The fourth phase, 'D distribution', is as follows: For a 'Reshelve' or successful 'Borrow' event, each 'D distributor' sends a message to the 'D update' process determined by the 'user', modulo 32. (The 'D distributor' for an event is always the same processor as its 'C collector' so no messages have to be passed between the third and fourth phases.) For 'Buy' events and unsuccessful 'Borrow' events, no message is sent (or needed). For an 'Audit' event, its 'C collector' broadcasts to each 'D' process a message containing the number of books in stock. It is therefore similar to the 'C distribution' phase.

The fifth phase, called the 'D update', is essentially similar to the 'C update'. It is followed by the sixth, 'D collection', phase, which is similar to the 'C collection' phase, and which computes the total number of books on loan.

Following the sixth phase, each processor then displays the results of its 'Audit' events. The time required to write these outputs is excluded from measurement. This is because all output on the CM-5 has to pass through the bottleneck of the single host processor.

Actually, the treatment of the 'Audit' event is more general than this particular problem demands. The '{C}' and '{D}' processes are allocated to the same 32 physical processors, so that each '{D}' process already has a list of all the 'Audit' events, because its processor has just finished executing a '{C}' process. Furthermore, the coordinating processor allocated to an 'Audit' event in the 'D collection' phase is the same one allocated in the 'C collection' phase, so it already knows the number of books in stock. It is therefore completely unnecessary for 'Audit' events to be broadcast to the '{D}' processes at all. However, for global events of a more general type it is likely that the processing during the 'D update' phase would depend on the result of the 'C update' phase — as it does for 'Borrow' events, so a broadcast would be needed. Including the broadcast was therefore considered a fairer measure of speed up.

A *C* program that implements this scheme can be found in [Dwyer 1995].

3.6 Analysis of the Parallel Update Algorithm

It is not enough to demonstrate that a parallel algorithm can be written, it is important to verify that it achieves a useful speed up. Unlike the sequential update algorithm, there is no hardware-independent heuristic for determining the point at which the parallel implementation breaks even with a single-thread benchmark. Instead, the break-even point depends on the ratio between update time and message-passing latency.

For an update event in the benchmark, the time per update is typically dominated by the time needed to read and write a record, the cost of two accesses. For a parallel implementation with P processors, it is necessary to add the cost of message passing and the time taken to synchronise the processors. Each update requires a message to be passed from a distributor process to an update process. The message needs to be acknowledged, so there are 2 messages in all. Since there are three phases in each update, three synchronisation steps are needed.

Suppose that one access takes time A, message latency is L, and synchronisation takes time S. Assuming that each processor takes an exactly equal share of the load, the break-even point is given by solving the following equation for U, where U is the number of events in a batch.

$$2AU = 2U(A+L)/P + 3S$$

If the number of processors is sufficiently large and the records are stored on disk, message latency is negligible compared with access time, so the equation simplifies to

$$2AU = 3S.$$

With any reasonable values of A and S, the resulting value of U is less than one. This is clearly a silly result, because one update cannot be distributed over several processors, but it shows that only a few updates are needed to break even.

What happens if the access time does not dominate the message latency? This could happen if the arrays are stored in main memory. In this case, access time is negligible compared with message latency, and the outcome depends on whether L/P exceeds A. If it does, no break even is possible. The only answer is to increase P, the number of physical processors. Otherwise, the time saved per update is $A-L/P$, and the break-even point is given by

$$2U(A-L/P) = 3S.$$

An experiment was conducted to test this prediction using the CM-5. The parallel algorithm was evaluated by timing its execution under a number of different experimental conditions. Details of the experiment are given in [Dwyer 1995].

The experimentally measured values of A, L and S are about 2μs, 30μs and 200μs respectively. With P=32, the expected break-even point is about 300 updates. In practice, the results depend on the size of the array that is updated. Because of caching effects, the access time for a larger array is greater than for a smaller array.

Figure 3.6.1 compares the actual performance of the parallel and benchmark algorithms graphically. The curve 'P6' shows the time per update for the parallel algorithm, when the 'C' and 'D' arrays each contain 1,000,000 elements. 'P2' shows the time per update for the parallel algorithm, when the 'C' and 'D' arrays each contain 100 elements. Curves 'B6' and 'B2' show the corresponding times for the benchmark algorithm. It will be seen that over 1,000 updates are needed for the parallel algorithm to amortise its overhead sufficiently to break even with the benchmark when the arrays are small, but only about 400 are needed when the arrays are large.

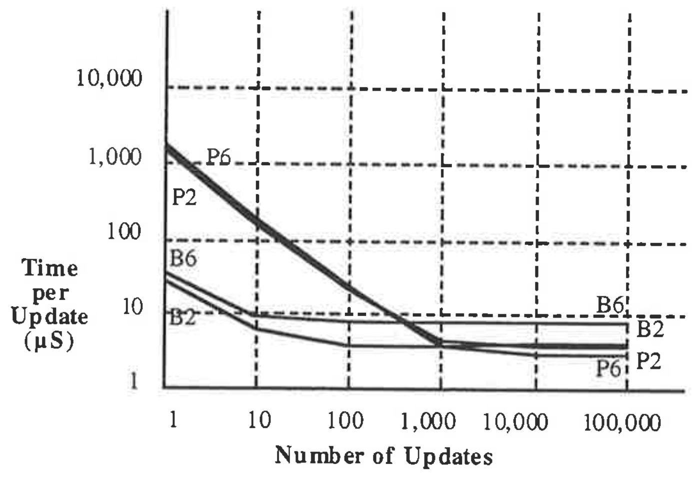

FIGURE 3.6.1: PARALLEL UPDATING COMPARED WITH THE BENCHMARK

Given that caching effects were significant, is it possible that accessing the arrays sequentially would improve the cache hit rate, leading to greater speed even when there is no disk input-output to consider? The short answer is 'No'. The snag is that the read loops in the 'C update' and 'D update' phases have to be made more complex, like those in the Cobol examples. Their extra complexity easily outweighs any advantage that results from more orderly access to the arrays.

Because 'Audit' events examine entire arrays, we should expect a speed up whenever the saving per array element exceeds the synchronisation overhead. By a similar argument to the update case, we obtain the following break-even formula,

$$N(A-L/P) = 3S.$$

where N is the number of array elements. (The factor of 2 is not present here because the array is inspected, but not updated.) The predicted break-even point occurs when N=600. Of course, if there are several 'Audit' events in a batch, the break-even should occur for smaller array sizes. On the other hand, when the number of queries is very small, the assumption that each processor is equally loaded must break down, because the distribution and collection phases of 'Audit' events are local to one processor.

Figure 3.6.2 compares the actual performance of the parallel and benchmark algorithms graphically. Curve 'P' shows the times per 'Audit' per element for the parallel algorithm, for a batch of 10 transactions, as the number of array elements is varied from 100 to 1,000,000. Curve 'B' shows the comparable curve for the benchmark algorithm.

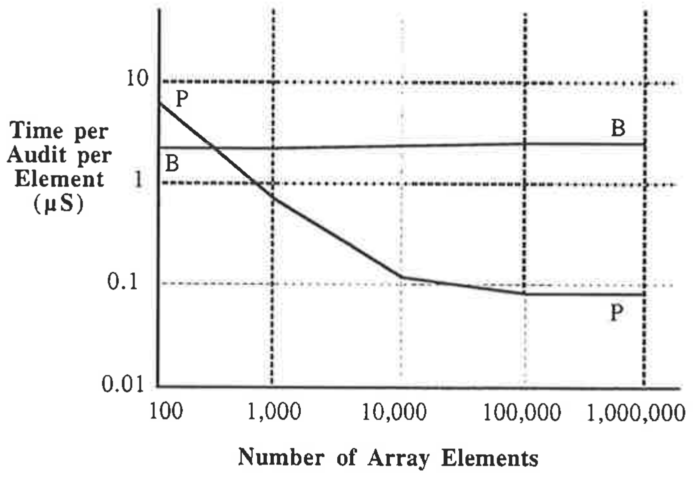

FIGURE 3.6.2: PARALLEL QUERIES COMPARED WITH THE BENCHMARK

What would happen if the events had been presented to the system as a single queue? It is not hard to predict. The distribution phase would have a single sending processor, so that the time to send and acknowledge each update event would be about 60μS, compared with 2μS when the latency is amortised over 32 processors. This would increase the time per update by a factor of 10–20, and in no case would the parallel algorithm offer a speed up greater than one. The situation for an 'Audit' is similar, except that a message has to be sent to every processor, increasing the time per 'Audit' by about 2,000μS. However, this overhead can be amortised over many array elements, so that a speed up is still possible for more than 1,000 titles or users.

Of course, if the arrays were stored on disk, access time would then dominate message passing, so starting with a single queue would not matter.

A defect of the experimental situation is that a real-world application is likely to need a persistent database, so that the arrays should be either stored or mirrored in disk files. This would always favour a parallel implementation, because disk access time (about 20mS) easily exceeds the latency of message passing (about 30μS). Unfortunately, the CM-5 handles disk input-output centrally, passing all data through the message network to a front-end processor. Disk input-output would become a bottleneck that made parallelism irrelevant, so no direct experiment could be done. If the CM-5 had had disk drives attached to each processor, an excellent speed up would have been obtainable. It was only possible to test the situation by using an experimentally measured time delay to simulate disk access.

Although with current technology, one would expect access times of the order of 20mS, the experimentally measured time per disk access on the CM-5 was about 2mS, irrespective of whether the access mode was random or sequential. Such a low average access time resulted because files were stored on a disk server with a very large internal cache. Most accesses involved only the cache. The server was connected to the CM–5 by an Ethernet network, so 2mS is really a measure of the latency of the network. Given that an average update needs about 3 disk accesses, the resulting 6mS overhead is easily enough to swamp the overhead of message passing. As might be expected, the simulated speed up from using 32 processors was close to 32 under a wide range of conditions — *giving a throughput of about 5,000 updates per second.*

Despite the random and sequential access times being equal, sequential access still has the advantage that no record is inspected or updated more than once. Given enough events in a batch — especially ‘Audits’ — the speed-up due to sequential access could be made be as great as desired. For example, if each record is inspected by an average of 3 events, sequential access is 3 times faster than random access. Combined with the speed up due to parallelism, sequential access resulted in simulated speed-ups exceeding 100, or about 15,000 updates per second.

Another possibility relevant to high-volume transaction processing is to use a main memory database [Eich 1992, Garcia-Molina & Salem 1992]. This technique stores the complete database in main memory, but also eventually records a persistent copy on disk. This means that no disk input-output is needed for queries, but updates must be recorded on a log file. Log records may be buffered in a small non-volatile memory, so logging can occur lazily, a disk access being required only when a buffer is full. This is a very efficient option, which becomes more attractive if each processor stores a partition of the database in its own persistent disk store. It is difficult to estimate the performance of this approach, because its speed ultimately depends on how often the log buffer needs to be emptied. Pessimistically, we may assume that a log record is written after each update to the database — an average of 1.5 disk accesses, or 30mS per update. Spread over 32 processors, the time per update would be about 1mS, or 1,000 events per second. More realistically, assuming that the log buffer can hold at least 10 updates, *it should be possible for 32 processors to easily exceed a rate of 10,000 updates per second.* For

1,000,000 elements, an 'Audit' event takes nearly 80mS — surprisingly slow compared with an update.

Sequential access and parallel processing are independent ways of improving performance, and can be used separately or in combination. But both hinge on the structure of the system specification in the same way. Neither sequential access nor parallel access is possible unless the accesses to the 'C' and 'D' arrays can be separated. But once that separation is made, all their elements can be processed independently.

Suppose the separation of the '{C}' and '{D}' processes had not been made. Would some other form of parallel access be possible? Consider first the possibility that the processes are parallel with respect to the 'C' array, but access the 'D' array randomly. Then, there is no guarantee that the history of a 'D' element will be correct. It is possible that it could be updated by a message sent from a fast running '{C}' process before a logically earlier update message arrived from a slower running '{C}' process. A similar objection applies to making the processes parallel with respect to the 'D' array. A third possibility is for each processor to process the events initially allocated to it. This is open to both objections.

Of course, correct results could still be obtained if some protocol were used to ensure that updates occurred in the correct order. There are many ways this could be done. The most familiar is the two-phase locking protocol used by many database management systems [Bernstein *et al.* 1986]. For example, each processor could process the events initially allocated to it, locking the 'C' and 'D' array elements it accesses. Locking an element would cause any other update that wanted to access the same element to wait until the lock was released. This would be done at the completion of each update procedure. (By always locking 'C' elements before 'D' elements, we guarantee that the system will not deadlock.) As it stands, this does not guarantee real-time equivalence, because a *later* event may cause the procedure for an *earlier* event to wait if they happen to access the same element. However, the system satisfies the serialisability criterion, the state of the system is consistent with some order of events, although the particular order cannot be predicted.

Even after relaxing the real-time equivalence requirement, the system would be no faster than the algorithm given here, because each access to a 'C' or 'D' element would require a message to be sent from the client process to the server process controlling access to the element, and back again. There would be no less message traffic, and indeed, there could easily be more: a message to read the value of the element, a second message to update it, and a third to release the lock. The alternative, which is to package reading and updating within a single procedure, saves a message, but it is essentially the solution presented here. In addition, two-phase locking can introduce further performance degradation through data contention thrashing [Wang *et al.* 1997].

'Audit' events complicate the issue further. It is not possible to update a shared database during an 'Audit' event. 'Borrow' and 'Reshelve' events cause changes to both the 'C' and 'D' arrays. Such an update that occurred when an 'Audit' had completed counting the stock but not yet counted the loans would be reflected in the loans, but not in the stock. It is therefore necessary for an ‘Audit’ to lock the entire database. This would effectively divide execution into update phases and query phases. In effect, the updates between each query would constitute a batch. If 1% of all events were ‘Audits’, the average batch would contain only 100 events. As discussed earlier, such small batches can lead to less than break-even performance.

So far, we have considered the possibility that the database is partitioned between the processors. An alternative is for it to be replicated. Suppose each processor has a copy of the whole of the ‘C’ and ‘D’ arrays. This certainly simplifies the ‘Audit’ events, which require no message passing at all. But now, each update must broadcast every change to a ‘C’ or ‘D’ element to every processor. This makes the processing of updates very slow. There is also a problem keeping the copies of the database consistent. The updates to each replica must be made in a consistent order. However, if the ratio of queries to updates is high, this is clearly an optimal arrangement. Conversely, when the ratio of queries to updates is low, a partitioned database is better. For intermediate ratios, it is possible for a mixture of replication and partitioning to work best; for example, there could be 4 copies of the database, each distributed over 8 processors.

Finally, we must make an important point about absolute time scales. Given that useful speed ups are obtained for batches of 1,000 records, and that throughputs can easily exceed 1,000 events per second, processing an entire batch can take less than one second of real time. This time is short enough to be considered virtually instantaneous in a typical transaction processing environment, so that from the user’s point of view, a parallel batch system might be indistinguishable from an on-line system.

3.7 Composite Keys

When a file has a composite key (modelled by an array with more than one dimension) the structure of Figure 3.4.1 may elaborated further. For example, suppose the library database recorded a file of loans, whose key was the combination of ‘title’ and ‘user’. There could be an update process for each ‘title’, and within it there could be an element update for each (title, user) combination. Figure 3.4.1 would then be both a diagram of the system as a whole, and also of each ‘title’ process. The structure could be elaborated in this way for as many component keys as desired.

In the discussion of sequential access in Section 3.3, it was pointed out that global events such as ‘Audit’ need to be stored in main memory throughout an update, allowing only a limited number to be present in a batch. On the other hand, updates to individual records are only transiently present in memory, and a batch can contain any number of them. A composite key opens the possibility of a third category: updates or queries to all the loans for a given title. Their event records would have to be buffered in main memory throughout the processing of the relevant title, so their number is limited, but the total number present in a batch may be much greater, because the buffer can be reused for each title.

Section 3.4 discussed (and rejected) the possibility of combining sequential access to one file with random access to a second file. We have also seen that similar objections apply to parallel algorithms. However, given the higher speed of parallel or sequential implementations, it is tempting to try to apply them more generally.

An opportunity arises when *two* master files are updated, one having a composite key, and the second having a key that is part of the key of the first. For example, in a parallel update, it would be possible to partition access to the loans file by 'title', by 'user', or by both. Suppose it were partitioned by 'title'. It would then be possible to update or inspect all the loans for one title within a single processor. Furthermore, the *same* processor could access the number of copies of the book, so that, for example, a modified 'Loan' event procedure could record the loan and decrement the number of copies within the same processor.

The analogous situation for sequential access is store the loans file in (title, user) sequence. The event records would then be sorted by 'title', then time — *not* (title, user) then time. This would allow concurrent sequential access to the books file, with the events for each book being processed in time order. The loans file would still need to be accessed randomly, because the updates for all loans involving a given title must be processed in time order, not 'user' order. However, access to the loans would not be entirely random, since a cluster of records with the same title would be accessed as a group. This would lead to more effective caching and faster access, especially if a typical group fitted within a single disk block. Although such a mixed mode update might prove slower than two separate sequential updates, it would certainly prove faster than a simple random update.

In general, we may conclude that whenever a set of files have keys that share a common component or components (or when a set of arrays share common dimensions) it is possible to exploit this, and achieve better efficiency than if they were processed by a simple read loop. Mixed mode access is optimum in the case of parallel access, because it reduces message traffic. However, in the case of sequential processing, although mixed mode access is better than random access, separate sequential updates are better still — provided the two update processes are separable.

3.8 Limitations

The update algorithms presented here lack a feature that is common in practice, that is the ability to report on the updating process itself. For example, a company might produce monthly statements of account by storing all events for a month, then updating customer balances. The statements of account are simply by-products of the update. Rather than in response to an input event requesting them, the statements are produced every time the update program is executed.

It is not possible to specify such reports using the Canonical Decomposition Method without first introducing the idea of a meta-event: an event about events. Instead of adding this new feature to the specification language, it is easier to model such reports indirectly. Essentially, the specification must be amended in one of two ways. The first is for the updates to be recorded in the database as they occur, without acting on them. The production of the statements of account then becomes an event that reports these events while updating customer balances. The drawback of this approach is that it transfers the task of specifying the whole system to that of specifying this event, effectively requiring the specifier to solve the system design problem. An easier and more flexible approach is for the updates to be acted on as they occur, and to record their effects in the database. The report event then consumes these records.

Unfortunately, it is then impossible to recognise that the effects of the updates could be recorded transiently in primary storage instead of the database. This is because it is not possible to specify that the report event is always synchronised with the processing of each batch. If it is, then the stored set of results is empty at the start and end of each batch, so it doesn't need to be stored persistently. On the other hand, if this condition is not guaranteed, for example, if updates are processed daily and reports are produced monthly, it is essential to store the results between batches. The limitations of the preceding update algorithms and specification language make it necessary to implement the more general situation in all cases. As a result, a useful opportunity for optimisation may be lost. However, it seems reasonable that a future extension of the specification language could solve the problem.

A second limitation concerns giving priorities to events. For example, if a customer changes a postal address, this event may be given a higher priority than earlier sales made to the customer. This is because any invoices, delivery notes, etc., that are printed for the customer should be delivered to the new address rather than the old one. In consequence, many update programs sort events by their types as well as by their time-stamps, to give priority to events such as these.

This limitation is not a serious one. If an event such as a change of address is supposed to have priority over earlier sales, it can simply be given an artificial time stamp that places it ahead of them. The system state should be updated according to the times when events are supposed to be effective, rather than when they happen to be recorded. Adjusting time stamps is a much more flexible scheme than assigning each kind of event a fixed priority.

If input events are presented to the system in batches, e.g., as files, their timing information may be implicit. For example, they may be implicitly time-stamped for the current date. In such a case the time stamp may sometimes be too coarse grained to distinguish the relative timing of events within the same batch. A familiar example is provided by the way a bank typically handles a personal cheque account. A deposit followed by a withdrawal, both on the same day, will have the same date, but it may be a matter of chance which is processed first. If the withdrawal is erroneously processed before the deposit, it could appear that the account is overdrawn when it isn't. For this reason, banks often process all deposits before all withdrawals for the same date. In effect, deposits have earlier time stamps than withdrawals.

A more general and more flexible method is to assume that the order of events is their stored order, i.e., determined by the sequence of the event transfer file, so that each event has a serial number. There is often little difference between explicit timing information and implicit timing information such as an event serial number, because the final state of a system often depends only on the sequence of input events, not their specific times. Although the input to a system is really a mapping from times to events, it is usually enough to treat it as a sequence of events. When time does matter, it may be treated like any other attribute of the event.

4. Separability

This chapter discusses 'separability', the property that allows a system to be decomposed into a network of component processes and queues. We say that two processes are 'separable' if data flows in only one direction between them. If two processes are separable, they may be connected by a queue.

4.1 Real-Time Equivalence

The model of system on which this analysis is based can be symbolised by a time line, as in Figure 4.1.1. The horizontal strip represents an idealised history of a system. Time passes from left to right. The system has state S_{i-1} until event E_i occurs, after which its state becomes S_i. After event E_{i+1} its state becomes S_{i+1}, and so on. The events that change the state are instantaneous; the system is always in a defined state. Moreover, the changes of state occur in real time, i.e., as soon as the real-world events that they model. There are several respects in which this model differs from a realisable implementation, therefore it is called the 'ideal real-time system'. Although we cannot build an ideal real-time system, a correct implementation should in certain respects be equivalent to the ideal. Thus, we call this notion of correctness 'real-time equivalence'.

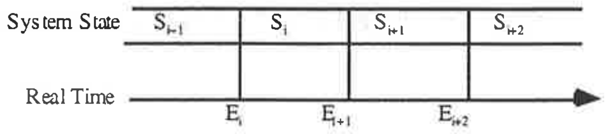

FIGURE 4.1.1: AN IDEAL SYSTEM TIME LINE

One respect in which an implementation may differ from its real time ideal is that changes of state may lag behind the real-world events they record. A second respect is that it must take a finite time for a state change to occur; during the change, the system state is undefined. This situation is symbolised in Figure 4.1.2. The white areas represent time intervals when the system state is well defined, the grey areas represent intervals when the state is changing, i.e., the new state is being computed from the old state.

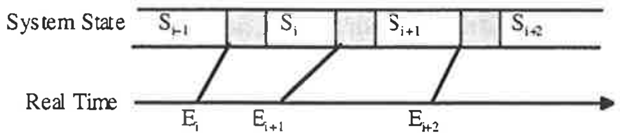

FIGURE 4.1.2: A MORE REALISTIC SYSTEM TIME LINE

It is important that the rate at which events arrive at the system does not exceed the speed with the system can respond. One possibility is to queue the input events so that the delay between the real event and the corresponding state change can be arbitrarily long. A second possibility, which is common in computer hardware, is for event E_{i+1} to be ignored during the response to event E_i, i.e., until the system is safely in state S_i. This means that the input is lost. A third possibility, which is common in human computer dialogue, is that the system only invites input when it is in a well defined state. Whichever of these strategies is used, events are always treated atomically and the response to one event is completed before the response to the next is begun.

Now consider a network of several component processes linked by data flows. The response of each component to an external event may be delayed differently. Assume that variable *u* is accessed by process *U*, and that variable *v* is accessed by process *V*. Let the values of *u* and *v* after event E_i be denoted by u_i and v_i.

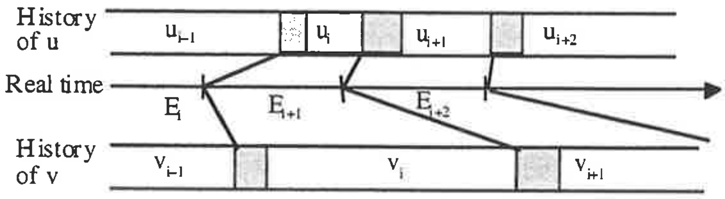

FIGURE 4.1.3: TWO SYSTEM COMPONENTS

Figure 4.1.3 sketches this situation. Its horizontal bars symbolise the histories of variables 'u' and 'v', with real time running from left to right. The white areas represent periods when the values of 'u' and 'v' are well defined, and the grey areas represent periods when new values are being computed. For example, the left-most upper grey rectangle of Figure 4.1.3 represents the period when u_i is being computed from u_{i-1}. At times the value of 'v' is more up-to-date than 'u', at others, 'u' is more up-to-date then 'v'. There are therefore times at which the system is in an inconsistent state. Indeed, it is possible that the values of 'u' and 'v' are never in step.

How can such a system be considered correct? The criterion used here is that the histories of the state variables (i.e., the sequences of their values) are the same as they would be in an ideal real-time system, although they may lag real time by arbitrary delays. The system implementation is then said to be 'real-time equivalent'. Although the state of a real-time equivalent system cannot be discovered by direct observation, it may be assembled by inspecting its variables at times that correspond to simultaneous points in their histories.

The treatment of temporary variables must be different from that of state variables. Temporary variables have short lifetimes, and exist only within the scope of one event. They do not have histories as state variables do. On the other hand, we must treat external packages similarly to state variables because we must ensure that outputs are generated in the correct order.

Real-time equivalence is one criterion of correctness. There are at least two other criteria that could have been chosen. One is that the histories of variables are unimportant, and that only the outputs of the system matter. This is an argument in favour of behaviour-based or sequence-based system description. However, it is hard to see how to guarantee correct behaviour except by enforcing some equivalent discipline on states. The second criterion is that of 'serialisability', used in multi-user database management. Serialisability requires that if events originate from different sources, the system should behave as though they originated from a single source. If two events from different sources do not interact, their relative ordering never matters; if they interact, either can be declared to have priority. However, the original ordering of events from a common source must always be respected. Serialisability and real-time equivalence may be reconciled by assuming that all events are time stamped, and that if two interact, their time stamps are used to order their priorities. In real-time equivalence, the time stamps are applied globally as events enter the system, but in serialisability they may be applied lazily, as the need arises.

The following sections will develop a theory of system design based on three basic ideas:

1. The notion of a system as a network of processes connected by one-way data flows or queues, modelled by delayed procedure calls.

2. The use of real-time equivalence as a criterion of correctness.

3. The use of rewriting rules to derive the specifications of component processes from the specification of the system as a whole.

4.2 Event Timing

The concept of delayed procedure call discussed in Section 2.3 is not itself enough to guarantee real-time equivalence. The supporting framework of message passing must preserve the order of events that reach each component process.

The simplest way to preserve event order is for each input event to be processed to completion before the next is accepted, but this prevents any benefit from being obtained from parallel processing. The next simplest arrangement is a pipeline, like that shown in Figure 1.4.2 of Page 7. Because there is only one path between any two processes, messages cannot overtake one another.

Some system topologies provide different pathways from the source of input to a given process, which cause different delays, allowing messages to overtake one another. Figure 1.5.2 showed how this can happen. Events reaching the '{D(1)}' process, say, can pass through any of '{C(1)}', '{C(2)}', or '{C(3)}'. Once there are multiple pathways between processes, it becomes essential for events to be time stamped, so that they can be processed in the right order. This requires that each process should carefully merge the input streams it receives to ensure that it processes each new event only after all earlier ones.

However, time stamps are not enough in themselves. If a process has received event messages for events E_i, and E_{i+1}, it remains uncertain that it can process event E_i, because an outstanding message for event E_{i-1} may still arrive by a slower pathway. One way to resolve this is to broadcast all events along all pathways. An event can then be safely acted on as soon as all earlier messages have been received from all possible sources. This is not an efficient solution; the bulk of message traffic consists merely of timing information.

An acceptable alternative is to broadcast timing messages only occasionally. When a timing message is received, it implies that no outstanding message can arrive over the path concerned. However, until a timing message is received, all incoming event messages must remain queued, in case some earlier event has yet to arrive by a slow pathway. The timing messages therefore divide the input events into batches. In an interactive system, timing messages should be frequent and batches will contain few events. In a non-interactive system, batches may contain all the events for a whole day, or even longer.

```
type user is private;
type book is private;
package Library is
   procedure Check (u: user);
end Library;
package body Library is
   D : array (user) of natural := (others => 0);
   E : array (user) of boolean := (others => false);
   procedure Check (u : user) is
   begin
      E(u) := true;
      if D(u) <= 10 then
         E(u) := false;
      end if;
   end Check;
end Library;
```

EXAMPLE 4.2.1: CHECKING FOR GREEDY BRANCHES IN A LIBRARY SYSTEM

It is possible for one input event to cause several delayed procedure calls. Example 4.2.1 shows how this might happen. The event 'Check' is intended to find those branches who have borrowed more than 10 books. (The result of the event is to set an element of 'E' to true or false.) The specification can be decomposed into separable '{D}' and '{E}' components, as in Example 4.2.2. The first component ('C') makes two calls on the second ('D'), the order of which must be preserved. The time stamp given to these internal calls must be more fine-grained than the time stamp given to the external event. A simple possibility is to use a Dewey number, so that, if the input event has time stamp 't', the delayed calls have time stamps 't.1' and 't.2'. This idea can be extended to several levels, e.g., 't.2.3.2'. Although in this example the numbering can be assigned statically, in general, e.g., when a **while** loop makes an unpredictable sequence of delayed calls, it must be assigned dynamically by counting the delayed procedure calls.

```
package body Library is
   package C is
      procedure Check (u : user);
   end C;
   package D is
      procedure Check1 (u : user);
      procedure Check2 (u : user);
   end D;
   package  body C is
      D : array (user) of natural := (others => 0);
      procedure Check (u : user) is
      begin
         D.Check1(u);
         if D(u) <= 10 then
            D.Check2(u);
         end  if;
      end Check;
   end C;
   package  body D is
      E : array (user) of boolean := (others => false);
      procedure Check1 (u : user) is
      begin
         E(u) := true;
      end Check1;
      procedure Check2 (u : user) is
      begin
         E(u) := false;
      end Check2;
   end D;
end Library;
```

EXAMPLE 4.2.2: CHECKING FOR GREEDY BRANCHES — COMPONENT PROCESSES

4.3 Timing of Separable Processes

This important section considers the conditions under which a real-time equivalent system can be decomposed into separable processes. The key concept is that connecting two processes by a queue means that the sending process can never lag behind the receiving process. Queues therefore induce a partial ordering on processes that controls their relative lags behind real time.

Consider the situation when two processes are connected by delayed procedure call. Suppose the calling process '{U}' has exclusive access to variable 'u', and the receiving process '{V}' has exclusive access to variable 'v'. Figure 4.3.1 shows possible histories of 'u' and 'v', including a situation where event 'E_i' causes 'u' to be updated. During the updating of 'u', a delayed procedure call is made to update 'v'. In Figure 4.3.1, the updating of 'v' is completed first, so that briefly, 'v' has value 'v_i', while 'u' is still undefined. Subsequent calls are delayed, so that the value of 'v' lags behind the value of 'u', so that for example, 'v_i' coexists with 'u_{i+2}'. This is possible because there are two delayed calls in the queue linking processes '{U}' and '{V}'. There is a delay of 2 events at this time between '{U}' and '{V}'. The delay need not be constant. It is a property of the queue linking the processes that the delay may be zero or arbitrarily great, although it may never be negative. When two processes may be linked by a queue, we say they are 'separable'. This property leads to a key theorem:

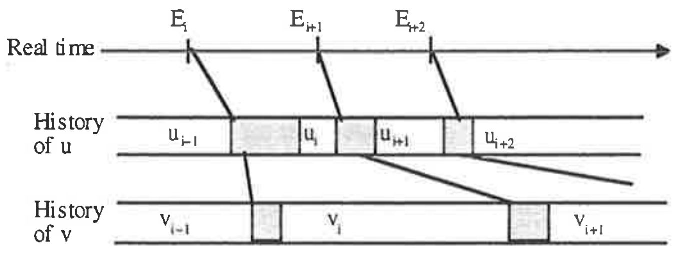

FIGURE 4.3.1: DELAYED PROCEDURE CALL

The Data Flow Theorem:

The queues connecting the separable processes of a system define a partial ordering.

Proof

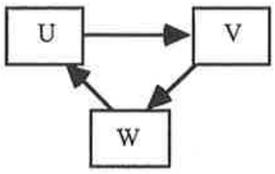

FIGURE 4.3.2: A CYCLE OF QUEUES

Draw a graph whose vertices are processes and whose directed edges are queues. Suppose that contrary to the hypothesis, this graph contains a cycle. For example, Figure 4.3.2 shows a situation where process '{U}' sends messages to process '{V}', which sends messages to process '{W}', which sends messages to '{U}', cyclically. Then the state of 'v' may lag arbitrarily behind the state of 'u', the state of 'w' may lag arbitrarily behind the state of 'v', and the state of 'u' may lag arbitrarily behind the state of 'w'. Therefore the state of 'u' may lag arbitrarily behind itself, which is clearly a contradiction, proving that no cycle can exist. Therefore the graph must be acyclic and defines a partial ordering. (This argument generalises to cycles of any length.)

It is not *wrong* for several processes to communicate cyclically, for example, for two processes to exchange the values of their variables. One way to model such communication is by remote procedure call. For any given event, several such calls might be in progress. However, for a process to bring its variables to their new states, each call it has made must complete, and this must occur before it can process the next event. Such data-flows cannot properly be called queues, as they must always contain a bounded number of messages and must always be empty between events; nor can the processes be called 'separable'.

Corollary 1:

A state variable may be accessed by at most one separable process. (On the other hand, there is no restriction that each separable process should access only one variable; it is easy to imagine a correct implementation in which a single process updates all the state variables.)

Proof:

In order to preserve the correct sequence of updates and inspections of the variable, it is necessary, in general, that different processes that inspect the variable have the same lag. This cannot be guaranteed if the processes are linked by queues.

This result is necessary to justify the rewriting rules that have already been presented, in particular, the idea that variables are private to component processes. Figure 1.4.2 (Page 7) gave an example where an erroneous attempt was made to share a variable between two separate processes. The problem in that example was that updates of the numbers of copies of books lagged behind their inspections, so that the inspected values could be out of date.

Corollary 2:

A set of separable processes can always be connected as a pipeline.

Proof:

Every partial ordering has at least one topological sort. (A partial ordering may be drawn as an acyclic graph. A topological sort of an acyclic graph is a sequence of its vertices such that, for any pair of vertices 'u' and 'v', if there is an edge from 'u' to 'v' in the graph, then 'u' precedes 'v' in the sequence.)

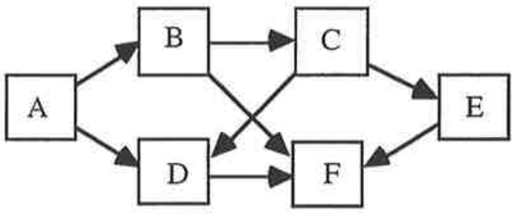

FIGURE 4.3.3: A PROCESS GRAPH

Figure 4.3.3 shows a possible graph of processes and queues. Figure 4.3.4 shows one of its two topological sorts. (The other places 'E' before 'D'.) In Figure 4.3.3, process '{F}' must merge inputs from processes '{B}' and '{D}'. In Figure 4.3.4, the path from '{B}' to '{F}' is implemented by passing calls transparently through processes '{C}' and '{D}', and no merge is needed. Given the earlier discussion of the importance of preserving the order of events, the pipe-line structure is usually easier to implement.

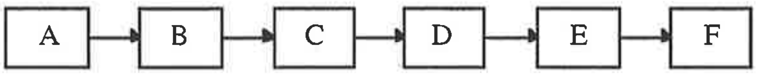

FIGURE 4.3.4: A PIPE-LINE

4.4 Dependence

Because the primary purpose of queues is to pass data between processes, this suggests that variables accessed by the downstream process depend on variables accessed by the upstream process. This section explores the relationship between variable dependences and queues.

If some event can cause the new state of 'v' to be conditional on the existing state of 'u', we say that 'v' **depends on** 'u'. That is to say, there is at least one situation in which it will be necessary to inspect the value of 'u' before updating 'v'. Consequently, the {U} process cannot follow the {V} process.

Suppose that the state of variable 'v' depends on the state of variable 'u'. (An example where 'D(u)' depends on 'C(t)' is provided by Figure 1.4.1 (Page 6).) By 'v depends on u' is meant that some event E_i uses the value of u_{i-1} to compute v_i. That is, v_i cannot be calculated until after the value of u_{i-1} has been inspected. Since this inspection occurs during the processing of event E_i, whenever 'u' and 'v' are well defined, 'u' never lags behind 'v'. However, the dependence puts no constraint on the lag of 'v' behind 'u'; old values of 'u' may be stored until needed.

The Dependence Theorem:

Dependences forbid queues: If process '{V}' accesses a variable 'v' that depends on a variable 'u' accessed by process '{U}', then there may not be a queue of delayed calls flowing from process '{V}' back to process '{U}'. (There may or may not be a queue from '{U}' to '{V}'.)

Proof:

If there were such a queue, then the state of 'u' could lag arbitrarily behind the state of 'v'. But 'u' cannot lag 'v', because 'v' depends on 'u'.

Unfortunately, given a set of event specifications, it is not always easy to decide where dependences exist. For example, Example 4.4.1 is a possible event procedure in which 'v' appears to depend on 'u', but is actually independent of it.

```
procedure Apparent_Dependence is
begin
   if u > 0 then
      v := 1;
   else
      v := 1;
   end if;
end Apparent_Dependence;
```

EXAMPLE 4.4.1: AN APPARENT DEPENDENCE

If a dependence of 'v' on 'u' exists, it may be shown by example; other variables remaining the same, it is merely necessary to find two different values of 'u' that result in two different values being assigned to 'v'. Proving that an apparent dependence does *not* exist must be done symbolically, by showing that the given specification could be replaced by one without an apparent dependence. This can be very difficult, because it is in general undecidable to show whether two programs are equivalent. It is easier to take a more pragmatic view, and consider that the way the procedure of Example 4.4.1 is defined *makes* ‘v’ depend on ‘u’. In order to execute the procedure as specified, it is necessary to inspect ‘u’ before updating ‘v’. If ‘v’ is to be considered independent of ‘u’, the procedure should have been written differently, e.g., as in Example 4.4.2.

```
procedure Apparent_Dependence is
begin
   v := 1;
end Apparent_Dependence;
```

EXAMPLE 4.4.2: THE DEPENDENCE REMOVED

Having a precise definition of dependence is unimportant when system design is done by a human. A human can recognise (or fail to recognise) the absurdity of Example 4.4.1, and correct it (or fail to correct it). But an automated system that will analyse specifications to derive dependences needs a precise definition of dependence. CDM derives a set of component processes for a system, and derives their specifications using the rewriting rules for delayed procedure calls. Therefore it must link its notion of dependence directly to its rewriting rules. Since it has no rewriting rule that can convert Example 4.4.1 into Example 4.4.2, it must consider ‘v’ to depend on ‘u’.

```
package body Process_U is
   procedure Apparent_Dependence is
   begin
      if u > 0 then
         Process_V.Apparent_Dependence_1;
      else
         Process_V.Apparent_Dependence_2;
      end if;
   end Apparent_Dependence;
end Process_U;
package body Process_V is
   procedure Apparent_Dependence_1 is
   begin
      v := 1;
   end Apparent_Dependence_1;
   procedure Apparent_Dependence_2 is
   begin
      v := 1;
   end Apparent_Dependence_2;
end Process_V;
```

EXAMPLE 4.4.3: APPARENT DEPENDENCE COMPONENTS

Example 4.4.3 shows a two-process implementation of Example 4.4.1, which is possible because ‘u’ does not depend on ‘v’. Example 4.4.3 may be transformed into Example 4.4.1 by the rewriting rules. However, an alternative decomposition in which process ‘{V}’ calls process ‘{U}’ is impossible within the rewriting rules. The dependence of ‘v’ on ‘u’ forbids a queue to flow from process ‘{V}’ to process ‘{U}’. The alternative decomposition would only be possible only if the rules were extended so that Example 4.4.2 could be derived first, removing the dependence of ‘v’ on ‘u’.

In this thesis, such extensions to the rewriting rules are seen as a side-issue. The transformation of Example 4.4.1 to Example 4.4.2 is seen as the responsibility of the specifier. The *Designer* program makes use only of the rewriting rules for delayed procedure call described in Section 2.7. This does not preclude a future version including additional rules to remove apparent dependencies.

```
procedure Local_Reuse is
   t: natural := 0;
begin
   t := A;
   B := t;
   t := C;
   D := t;
end Local_Reuse;
```

EXAMPLE 4.4.4: RE-USE OF A LOCAL VARIABLE

Another source of apparent dependencies is the re-use of variables. The procedure ‘Local_Reuse’ of Example 4.4.4 shows an apparent dependence that results from the re-use of the local variable ‘t’. (‘A’, ‘B’, ‘C’ and ‘D’ are state variables.) Since ‘t’ depends on ‘A’, and ‘D’ depends on ‘t’, it may be falsely concluded that ‘D’ depends on ‘A’. It is relatively easy to eliminate this kind of false dependence by distinguishing the two definitions of ‘t’. In its first definition (‘t := A’) ‘t’ depends only ‘A’, and in its second (‘t := C’) ‘t’ depends only on ‘C’. Since the definition of ‘D’ (‘D := t’) uses the second definition of ‘t’, ‘D’ depends on ‘C’, but not on ‘A’. Such use-definition analysis is a standard compiler optimisation technique, with low computational complexity.

```
package body AB is
   procedure Local_Reuse is
      t1: natural := 0;
   begin
      t1 := A;
      B := t1;
      Process_CD.False_Dependence;
   end Local_Reuse;
end AB;
package body CD is
   procedure Local_Reuse is
      t2: natural := 0;
   begin
      t2 := C;
      D := t2;
   end Local_Reuse;
end CD;
```

EXAMPLE 4.4.5: RE-USE OF A LOCAL VARIABLE: COMPONENTS

Unfortunately, even this simple case is an embarrassment to the rewriting rules of Section 2.7. Suppose the specification of ‘Local_Reuse’ is made into two processes, ‘{A,B}’ and

'{C,D}'. The declaration and use of 't' cannot be placed in one process alone. The two definitions of 't' must be declared separately, as in Example 4.4.5. The procedure must be transformed to make rewriting possible. Although the transformation is simple in this case, it is more subtle in Example 4.4.6, where a system variable is used as a temporary store. Despite appearances, neither does 'D' depend on 'A', nor 'B' on 'D'. However, the procedure can be implemented by an '{A,B}' process and a '{C,D}' process, for example, provided the first two occurrences of 'D' are replaced by occurrences of a new local variable.

```
procedure Global_Reuse is
begin
   D := A;
   B := D;
   D := C;
end Global_Reuse;
```

EXAMPLE 4.4.6: TEMPORARY USE OF A STATE VARIABLE

In view of these difficulties, the discussion will initially ignore the extra sophistication of use-definition analysis. The emphasis will be on clear and simple specifications that avoid false dependences. Such specifications are in any case easier to read. They also simplify the discussion. It is much easier to say "Variable 't' depends on variable 'A'", rather than "The definition of variable 't' in 't := A;' depends on the initial definition of variable 'A'", and so on. Basing dependences on variables rather than definitions of variables is a gross simplification that makes a very poor analysis of some specifications. In Chapter 8, it will be explained how the basic method may be extended to employ use-definition analysis.

4.5 A Working Definition of Dependence

This section defines dependence so that it dovetails with the rewriting rules. This ensures that the set of component process specifications can be derived directly from the system specification. Dependences will be used to derive the process network, and to allocate variables to component processes. Procedures in component processes can access only the variables allocated to their parent processes or the variables passed to them as input parameters. However, the rewriting rules further limit the ways these procedures may be constructed. What matters is that the notion of dependence is consistent with the rewriting rules for delayed procedure call.

The rewriting rules replace a procedure call by the statements contained in its body. A call is therefore equivalent to a group of whole statements, not statement fragments. So, if an outer statement encloses an inner statement, the outer statement may call a procedure containing the inner one, but the inner statement may never call the outer one. By defining dependence so that the variables appearing in the inner statement 'depend on' the variables appearing in the outer statement, the effect is to forbid a queue from flowing from the inner statement back to the outer one. The definition of dependence must ensure that if 'v' depends on 'u', the process accessing 'v' may never call a procedure in the process accessing 'u'. We therefore make the following rules:

1. In an assignment of the form 'v := E', 'v' depends on every variable appearing in 'E'. (This rule expresses the idea that 'E' can only be evaluated if the variables that appear in it are either accessed in the same procedure as 'v', or are passed to it as input parameters.)

2. In an assignment of the form 'v(i,j) := E', 'v' depends on 'i' and 'j' (with an obvious generalisation to any number of suffixes).
(This rule expresses the idea that the subscripts used to access a variable must either be accessed in the same procedure as the variable, or passed to it as input parameters. Where parallel processing is an option, the subscripts are used to select the instance of the process that is called, and the same conditions apply.)

3. If an assignment of the form 'v := E', occurs within a conditional statement, e.g., of the form '**if** B **then**... **end if**;', then 'v' depends on every variable appearing in 'B'. (By 'conditional statement' is meant an **if** statement or a **while** statement.)
(This rule is motivated by the fact that either the conditional statement and the assignment may appear in the same process, or the process containing the conditional statement may call the process containing the assignment. However, the assignment may never call a process containing the conditional statement that encloses it, because the rewriting rules allow only complete statements to be replaced by calls. Therefore the assignment would have to call a statement containing itself, which is absurd.)

4. If an assignment of the form 'v := E', occurs within a **for** statement of the form '**for** i **in**...' or an **all** statement of the form '**all** i **in**...', then 'v' depends on 'i'.
The **for** or **all** statement itself is treated as an assignment to the loop variable.
(This rule is motivated by the fact that either the **for** or **all** statement and the assignment may appear in the same process, or the process containing the loop may call the process containing the assignment. However, the assignment cannot call a process containing the loop that encloses it, because the rewriting rules allow only complete statements to be replaced by calls. Therefore the assignment would have to call a statement containing itself, which is absurd.)

The above rules are straightforward, and correspond to an intuitive understanding of dependence. However, a fifth rule must be introduced, with a different motive.

5. If any expression contains an indexed term 'v(i,j)', 'v' depends on 'i' and 'j' (with an obvious generalisation to any number of suffixes).

This fifth rule is contrary to intuition, because the simple occurrence of 'v' in an expression does not affect its value, so that no real dependence can be said to exist. The effect of the rule is to treat a reference to 'v' similarly to an assignment to 'v'. The dependence forces 'i' and 'j' to be accessible to the procedure that accesses 'v', either directly or as parameters.

Suppose state variable ‘v’ is accessed in process ‘{V}’ but ‘i’ and ‘j’ are accessed in a later process ‘{W}’, called by ‘{V}’. Then the whole array ‘v’ must be passed as a parameter of the call, so that the element ‘v(i,j)’ can be selected by ‘{W}’.

Unfortunately, a state variable can’t be passed by reference, but must be copied. This is because, by the time the called process inspects it, a later event reaching the calling process may have changed its value. To ensure that the value of the variable cannot be corrupted, it must be copied. Remembering that the array ‘v’ could be very large, e.g., implemented by a database table, passing its whole value would be very inefficient. It would dramatically increase the complexity of the event procedure concerned. A procedure that accesses an element of an array of N elements should be expected to have complexity $\mathbf{O}(\log N)$, whereas copying the whole array has complexity $\mathbf{O}(N)$. To remove this source of additional complexity, the parameters of event procedures may not be state array variables.

However, local arrays are different, and may be passed as parameters. There is a separate instance of a local array for each event, so no other event can corrupt it. Therefore they may be passed by reference, and don’t need to be copied.

4.6 State Dependence Graphs

A *state dependence graph*, or SDG, records the dependences between variables that arise from analysing a set of events. (It is sometimes convenient to consider single events, sometimes all the events that a system must process, or sometimes a subset of them.) It is an abstraction of the dependence information that can be distilled from the text of a system specification.

The vertices of an SDG represent the state variables, the parameters and local variables of events, and external packages, and are labelled accordingly. Since the parameters and local variables of different event procedures may have identical names, the labels of their vertices may be made unique by writing them in the form ‘E.v’, where ‘E’ is the name of the event procedure, and ‘v’ is the name of the local variable or parameter.

Strictly, every element of an array should be represented by a separate vertex. However, for most purposes it is enough to use one vertex to represent a whole array. (This convention assumes that the elements of an array have similar dependences.)

The edges of an SDG represent dependences between variables. Specifically, if ‘v’ depends on ‘u’, then a directed edge must be drawn *from* ‘u’ *to* ‘v’ — the direction of the edge matches the direction of the data flow. Because a system design must be valid for each event, the set of edges that result from several events must be the union of the edges for each event.

Although an SDG should have a vertex for every input, every state variable, every temporary variable, and every output, unless a special point is to be made, only the state variables are drawn, to keep the diagrams as simple as possible. The input variables are always sources and the output variables are always sinks, so they can be taken as read. Temporary variables can be omitted because they don't have to be assigned to processes; they are typically used to carry data between them. Drawing temporary variables sometimes makes an SDG simpler and sometimes not. A temporary variable can usually be regarded as standing for the expression assigned to it, and treated as the bundle of variables appearing in the expression. After an SDG has been drawn and analysed, it is easy to see where the missing vertices should be attached.

```
procedure Borrow (t : title; u : user) is
begin
   if C(t) > 1 then
      D(u) := D(u) + 1;
      C(t) := C(t) – 1;
   end if;
end Borrow;
```

EXAMPLE 4.6.1: A BORROW EVENT

Consider the 'Borrow' event procedure of Example 4.6.1. Figure 4.6.1 shows its corresponding SDG. 'C' depends only on 't' and itself, but 'D' depends on 'C', 't', 'u' and itself. The arrow drawn from 'C' to 'D' suggests the need for a data flow from 'C' to 'D'. It means that, other things being equal, it is possible for the flow to pass through a queue, or a 'C' process to use delayed call to communicate with a 'D' process. Conversely, there can never be a queue flowing from 'D' to 'C'.

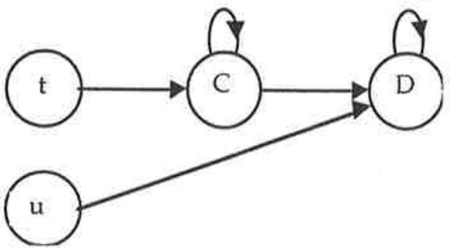

FIGURE 4.6.1: SDG FOR THE BORROW EVENT

Suppose the 'Borrow' event in the specification of Example 4.6.1 were revised to become that of Example 4.6.2. This change makes it impossible for any branch to borrow more than 10 books, introducing a new dependence of 'C' on 'D'. The SDG of the revised specification is given in Figure 4.6.2.

```
procedure Borrow (t : title; u : user) is
begin
   if D(u) < 10 and C(t) > 1 then
      D(u) := D(u) + 1;
      C(t) := C(t) – 1;
   end if;
end Borrow;
```

EXAMPLE 4.6.2: A REVISED BORROW EVENT

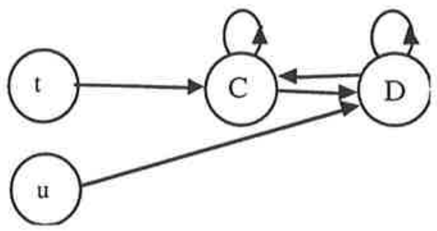

FIGURE 4.6.2: SDG FOR THE REVISED LIBRARY SYSTEM

Remembering the Dependence Theorem, i.e., that dependences forbid queues, the cycle between 'C' and 'D' suggests that data must flow in both directions between the '{C}' and '{D}' processes, and they can no longer be linked by a queue of delayed calls. Suppose that process '{C}' calls '{D}' as before. Where in the program text could the call be written? It could not merely replace the assignment to 'D'; 'D' must be accessed by the **if** statement in the calling process. Nor could it replace the whole **if** statement. Since the assignment to 'C' occurs within the **if** statement, the 'C' process could only update the value of 'C' if the called procedure passed the new value of 'C' back to the calling process as an output parameter. Delayed procedure calls forbid output parameters; data can't flow through a queue backwards. Since the rewriting rules only allow complete statements to be enclosed in procedures, these are the only two interesting options. But even ignoring the rewriting rules, a delayed call is clearly impossible, because 'C' depends on 'D' in the intuitive sense too.

If we suppose instead that process '{D}' calls process '{C}', an analogous argument proves that a delayed call is still impossible.

(Neither can access to the 'C' or 'D' arrays be shared by both processes; the two processes may be out of step, and the arrays have unique histories that cannot be synchronised with both processes at the same time. Nor does making two copies of 'C' or 'D' solve the problem.)

This argument does not prove that 'C' and 'D' cannot be accessed by physically remote processes, but only that their histories may never be allowed to become out of step (except transiently during the processing of an event). To process event E_i the values of '$C(t)_{i-1}$' and '$D(u)_{i-1}$' must both be available before '$C(t)_i$' and '$D(u)_i$' can be computed. The '{C}' and '{D}' processes must be closely coupled. The simplest option is to access both the 'C' and 'D' arrays in a single '{C,D}' process. We would only separate the processes if we were forced to by external considerations. For example, if 'C' and 'D' had to be stored at physically separate sites, we would have to take pains to ensure their states were kept in step.

```
package body C is
    C : array (book) of natural := (others => 0);
    procedure Borrow1 (t : title; C: out natural) is
    begin
        C := C(t);
    end Borrow1;
    procedure Borrow2 (t : title) is
    begin
        C(t) := C(t) – 1;
    end Borrow2;
end C;
package body D is
    D : array (user) of natural := (others => 0);
    procedure Borrow (t : title; u : user) is
        C: natural;
    begin
        D.Borrow1 (t, C);
        if D(u) < 10 and C > 1 then
            D(u) := D(u) + 1;
            C.Borrow2 (t);
        end if;
    end Borrow;
end D;
```

EXAMPLE 4.6.3: REMOTE PROCEDURE CALL

If for some reason the 'C' and 'D' arrays *must* be accessed in different processes, the communication can be modelled by 'remote procedure call', which does allow output parameters. Example 4.6.3 shows one model of close coupling. Component 'D' has access to the 'D' array. It makes two remote calls on 'C', which accesses the 'C' array. The first remote call returns an output parameter equal to the current value of 'C(t)'. The second remote call decrements 'C(t)'. Only the second remote call could be replaced by a delayed procedure call. In the following discussion, close coupling will be ignored. It involves the distributed database integrity problem [Lampson & Sturgis 1976, Gray 1981, Schneider & Lamport 1982], which is an issue in its own right, and outside the scope of this thesis. We shall say simply that 'C' and 'D' must be allocated to the same component process. The reader should remember that any non-trivial process can always be further decomposed into closely coupled parts.

Drawing dependences as edges of a graph suggests that paths in the graph may have important properties. For example, is dependence transitive? If 'v' depends on 'u' and 'w' depends on 'v' does that prove that 'w' depends on 'u'?

In themselves, dependences are not transitive. The fact that variable 'v' depends on 'u' and 'w' depends on 'v' does not prove that 'w' depends on 'u'. (An event specification may be simply 'w:=v; v:=u;'.) Despite this, because of the way dependence has been defined, the edges of SDG's *are* transitive.

An alternative reading of an edge from u to v is, according to the Dependence Theorem, that v may lag u, but u may not lag v. If v may lag u and w may lag v, then w may lag u. Conversely, if u may not lag v and v may not lag w, then u may not lag w. In terms of data-flows, if data flows from *u* to *v* and data flows from *v* to *w*, then data flows from *u* to *w*. In terms of queues, if no queue is allowed from *v* to *u*, and no queue is allowed from *w* to *v*, then no queue is allowed from *w* to *u*. We may therefore say that dependences are pseudo-transitive.

4.7 The Canonical Minimal Process Graph

There is obviously a strong connection between the SDG of a system and the process graphs of its possible implementations. For example, the edge from 'C' to 'D' Figure 4.6.1 suggests that a queue of delayed procedure calls could flow from a '{C}' process to a '{D}' process. (Which is why the dependence is drawn *from* 'C' *to* 'D' rather than the reverse.) More accurately, the edge makes a queue useful; it is the *absence* of an edge in the reverse direction that makes a queue possible. However, no queue is ever necessary. It remains possible to use a combined '{C,D}' process to implement the specification of Example 4.6.2 (although it may be less efficient). In the SDG of Figure 4.6.2, the cycle between 'C' and 'D' makes their processes strongly connected, and forbids the use of a queue. This is closely connected with the Data Flow Theorem, which forbids a cycle of queues in a process graph. What is the precise relationship between an SDG and a process graph?

Definition

The 'strongly-connected components' (or simply 'strong components') of a directed graph are maximal subsets of its vertices, such that 'u' and 'v' are members of the same strong component if the graph contains both a path from 'u' to 'v' and from 'v' to 'u', i.e., they form part of a cycle.

The strong components of a graph partition its vertices; each vertex is a member of exactly one strong component. Figure 4.7.1 shows the strong components of an arbitrary directed graph. Its components are {A,B,C}, {D}, {E}, and {F,G}. (Each component is enclosed inside an elliptical outline.) A component containing a single vertex, such as {D}, is called a trivial component. (Every vertex of a graph has a zero length path to itself.)

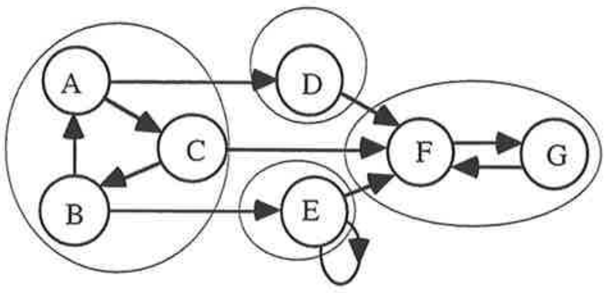

FIGURE 4.7.1: STRONG COMPONENTS OF A GRAPH

Definition:

The 'reduced strong component graph' (or simply the 'reduced graph') of a directed graph *G* has vertices that are the strong components of *G*. There is an edge from vertex

U to vertex *V* of the component graph if and only if there exist vertices *u* and *v* in *G* such that $u \in U$ and $v \in V$ and there is an edge from *u* to *v* in *G*.

Figure 4.7.2 shows the reduced graph of Figure 4.7.1. A reduced graph has the important property that it is always acyclic. Moreover, the reduced graph is connected if and only if the original graph is connected.

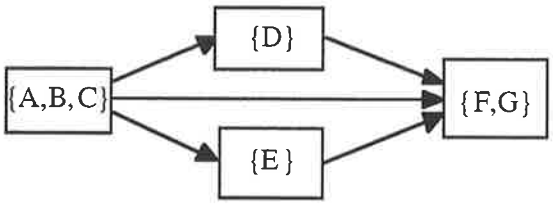

FIGURE 4.7.2: THE COMPONENT GRAPH OF FIGURE 4.7.1

Suppose that Figure 4.7.1 represents the SDG of some system. Then we claim that Figure 4.7.2 is a process graph of a network that can implement the system. It is also the most general process graph; every other feasible process graph can be derived from it. The vertices of Figure 4.7.2 represent its 'minimal separable processes'. A 'minimal separable process' corresponds to a set of variables that must be accessed either within a single process or by a group of closely coupled processes, but which could not be accessed by two or more processes linked by queues without losing the guarantee of real-time equivalence. The processes are 'minimal' in the sense that although they cannot be subdivided, they may be combined. For example, it would be possible to access 'E', 'F', and 'G' within a 'composite' process, giving the process graph of Figure 4.7.3. Finally, they are 'separable' in the sense that each vertex represents a process that can be linked by queues to the others.

(The vertices of an SDG, which represent variables, are drawn as circles. The vertices of process graphs, which represent processes, are drawn as rectangles. A component graph is a form of process graph.)

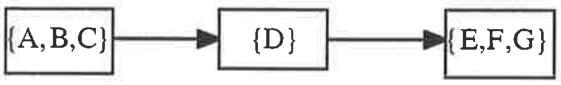

FIGURE 4.7.3: A PROCESS GRAPH DERIVED FROM FIGURE 4.7.2.

Figure 4.7.3 is one of several process graphs that can be derived from Figure 4.7.2. The 11 valid ways of composing the processes of Figure 4.7.2 are as follows:

{A,B,C},{D},{E},{F,G}.
{A,B,C},{D},{E,F,G}.
{A,B,C},{E},{D,F,G}.
{A,B,C},{D,E},{F,G}.
{A,B,C,D},{E},{F,G}.
{A,B,C,E},{D},{F,G}.
{A,B,C},{D,E,F,G}.
{A,B,C,D},{E,F,G}.
{A,B,C,E},{D,F,G}.
{A,B,C,D,E},{F,G}.
{A,B,C,D,E,F,G}.

But the remaining 4 possible compositions are invalid, such as {A,B,C,F,G},{D},{E} shown in Figure 4.7.4, because the pairs of edges joining {D} and {E} to {A,B,C,F,G} form cycles, and a cycle of queues is forbidden. Choosing the best valid composition is a potentially complex combinatorial problem, discussed in Chapter 9.

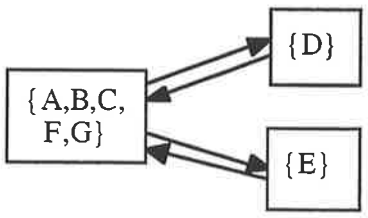

FIGURE 4.7.4: AN INVALID COMPOSITION BASED ON FIGURE 4.7.2.

In drawing various kinds of graph, such as SDG's, it is easy for them to become cluttered by numerous edges. If we are only interested in the existence of paths between vertices, it is possible to delete many of the edges and still retain the essential path information. This makes the drawing of a graph easier to understand.

Definition:

A 'transitive reduction' *H* of a graph *G* has the same transitive closure as *G* and contains the same vertices as *G*, but fewer edges. That is, if there is a path from *u* to *v* in *G*, there is also a path from *u* to *v* in *H*. Figure 4.7.5 shows the only transitive reduction of the graph of Figure 4.7.2.

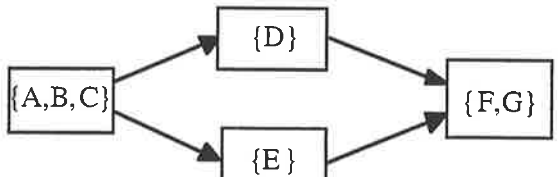

FIGURE 4.7.5: THE TRANSITIVE ROOT OF FIGURE 4.7.2.

Definition:

A 'minimal transitive reduction' *H* of *G* is a transitive reduction of *G* with fewest edges, i.e., such that no other transitive reduction of *G* has fewer edges than *H*.

The minimal transitive reduction of an acyclic directed graph is unique, and is called its 'transitive root'. It may be discovered by constructing a graph $G.G^+$ containing the vertices of G, and an edge for every composite path in G (i.e., paths of length 2 or greater). The edges of G that correspond to edges in $G.G^+$ are then removed [Aho *et al.* 1972]. (If G is the graph of Figure 4.7.2, the only edge in $G.G^+$ is from {A,B,C} to {F,G}. Figure 4.7.5 is the result of removing this edge.)

By finding its the strong components, and finding the transitive root of the reduced component graph, any directed graph may be transformed to a canonical acyclic form.

If Figure 4.7.1 is considered as an SDG, Figure 4.7.5 is called its canonical process graph, or CPG. The CPG is defines a partial ordering that constrains all possible process graphs that could implement any specification from which Figure 4.7.1 might be derived. Specifically, any system implementation must preserve the partial ordering of Figure 4.7.5. For example, Figure 4.7.5 is itself a valid process graph. So is Figure 4.7.3. So is any topological sort of Figure 4.7.5, e.g., the pipe-line of Figure 4.7.6. However, no valid process graph can include a queue that flows from one process to another that precedes it in the partial ordering. For example, no queue may flow from {D} to {A,B,C} of Figure 4.7.3 — which is precisely why the composition of Figure 4.7.4 is wrong. However, a queue may flow between {D} and {E}, because they are unordered.

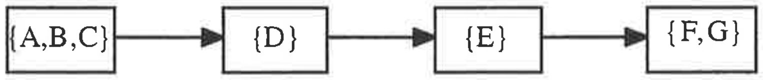

FIGURE 4.7.6: A TOPOLOGICAL SORT OF FIGURE 4.7.5.

If an SDG is connected, so is its CPG. We should certainly expect both graphs to be connected, because otherwise their connected regions describe independent systems. (Any event that accesses variables in more than one region could be split into parts that affected each region independently.) In the following discussion, it will generally be assumed that the graphs are connected. However, it makes no great difference if they aren't.

Definition:

A 'composite process graph' P is a 'valid composition' of a canonical process graph C if the vertices of P are labelled with sets of vertices of C, P is acyclic, and the edges of P are such that whenever there is a path from vertex u to vertex v in C, there is a path from vertex U to vertex V in P, where $u \in U$ and $v \in V$.

(There are therefore *two* stages of composition needed to derive a composite process graph from an SDG. First, state variables are grouped into strong components to form the vertices of the CPG, then selected vertices of the CPG are merged to form the composite process graph.)

It is now appropriate to demonstrate some of the claims of this chapter more formally. We first show that a reduced component graph is always acyclic.

Lemma 1:

Every reduced strong component graph is acyclic.

Proof:

Suppose that the reduced graph of G contains a cycle between distinct vertices U and V. Since the edges of the reduced graph result from edges in G, there must be a path from u_1 to v_1 in G, where $u_1 \in U$ and $v_1 \in V$. Likewise there must be vertices $v_2 \in V$ and $u_2 \in U$ such that there is a path from v_2 to u_2 in G. The vertices v_1 and v_2 need not be identical, but they must be strongly connected, because they are both members of V. Similarly, u_1 and u_2 must be either identical or strongly connected. Therefore there is a cycle passing through u_1, v_1, v_2, and u_2 in G. Therefore u_1, v_1, v_2, and u_2 are members of the same strong component of G, and U and V cannot be distinct vertices, contrary to the hypothesis.

Lemma 2:

The reduced strong component graph of G is connected if and only if G is connected.

Proof:

Let U and V be vertices of the component graph of G. Suppose that there is an edge from u to v in G, where $u \in U$ and $v \in V$. Then, because of the way it is constructed, there is a corresponding edge from U to V in the component graph of G. Conversely, if there is an edge from U to V in the component graph of G, there must be a least one pair of vertices u and v in G such that $u \in U$ and $v \in V$, and there is an edge from u to v in G. By induction, a pair of component vertices is connected if and only if two of their elements are connected. Therefore, every pair of vertices in the component graph of G is connected if and only if G is connected.

Lemma 3:

The transitive root of an acyclic graph is unique and acyclic.

Proof:

Given an acyclic graph G, compute the graph $G.G^+$ containing all composite paths in G (i.e., all paths of length greater than 1). Remove the corresponding edges of G. The resulting graph H is the transitive root of G. The process is deterministic, so H is unique. To see that H has the same closure as G, observe that the edges removed from G correspond to composite paths in H. Therefore they are generated by the transitive closure of H, so G cannot generate any edges in its closure that are not in the closure of H. To see that the transitive reduction computed in this way is minimal, observe that it contains no edges between vertices that are otherwise linked by composite paths. If any edge were removed from H, the edge could not be generated in the transitive closure of the modified graph, and it would not be a valid reduction of G. Since G is acyclic, and its transitive reduction is obtained by removing edges from G, its reduction is also acyclic.

The Separability Theorem:

Every state dependence graph has a unique corresponding canonical process graph that represents a feasible system design.

Proof:

Given a state dependence graph, its reduced strong component graph is obtained by a deterministic construction and is therefore unique. From the reduced strong component graph, the canonical process graph is obtained by constructing its transitive root. Any reduced strong component graph is acyclic, therefore its transitive root is acyclic and unique. Since the resulting CPG is acyclic, it cannot contain a cycle of queues. Therefore it represents a feasible system design.

Corollary 1:

The canonical process graph contains the maximal number of separable processes.

Proof:

The processes of the CPG correspond to the vertices of the reduced strong component graph of the state dependence graph. Since members of the same strong component are cyclically connected, placing any of its members in separate processes would result in a cycle of queues, which is forbidden by the Data Flow Theorem.

Corollary 2:

The CPG defines a partial ordering of its processes.

Proof:

Any transitive acyclic graph defines a partial ordering. The CPG is transitive either in the sense that an edge from U to V allows V to lag U, or in the negative sense that a queue cannot flow from V to U. Both these properties are transitive.

Corollary 3:

Every state dependence graph has at least one valid composite process graph, and at least one valid pipe-line process graph.

Proof:

Construct the CPG from the SDG. First, it is itself a possible composite process graph. Second, since the CPG defines a partial ordering, it has at least one topological sort. Connect the terms of the sort by queues to form a pipeline. (*See* Corollary 1.) Third, composing all the processes of the canonical graph into a single process is also a valid process graph. (The three cases are not necessarily distinct.)

(A simple way to find a topological sort is to choose any source vertex of the canonical minimal process graph (i.e., one that does not have edges entering it), and place it first in the pipeline. Delete the chosen vertex and its out-edges from the canonical minimal process graph. Now find a source vertex in the remaining graph, and place it second in the pipeline. Delete the vertex and its out-edges as before, and repeat until no vertices remain.)

4.8 Finding the Canonical Process Graph

This section explains efficient algorithms for finding the strongly connected components of a directed graph and a topological sort of an acyclic graph. These algorithms could form part of a computer program that could derive CPG's from specifications. The main point of the section is to show that there are polynomial-time algorithms that can derive a CPG from an SDG. In fact, the *same* algorithm can derive both the strongly connected components and the topological sort.

Depth-first search is a particular recursive procedure for exploring directed graphs. The pseudo-code procedure 'DFS' in Example 4.8.1 outlines depth-first search. 'Preorder' and 'Postorder' are arbitrary procedures that can be invoked during the search. 'Visited' is a boolean attribute of each vertex that is initially false.

```
procedure DFS (v : vertex) is
begin
   if not Visited(v) then
      Visited(v) := true;
      Preorder(v);
      for all edges v→w leaving 'v' loop
         DFS(w);
      end loop;
      Postorder(v);
   end if;
end DFS;
```

EXAMPLE 4.8.1: DEPTH FIRST SEARCH

As given in Example 4.8.1, 'DFS' is a procedure for making a depth-first search from one vertex. The vertices it visits, and the first edges followed that reach them, form a tree with the starting vertex as its root, called its 'depth-first search tree'. The depth-first search from one vertex does not necessarily visit all the vertices of the graph. A depth-first search of a whole graph may be made by augmenting the graph with a dummy source vertex with an out-edge leading to each true vertex of the graph, then making a depth-first search starting from the dummy vertex. The same effect may also be achieved by invoking 'DFS' for each vertex of the graph in turn, in any order, without reinitialising the values of 'Visited'.

The complexity of 'DFS' applied to a whole graph is $\mathbf{O}(|V|+|E|)$, where $|V|$ is the number of vertices, and $|E|$ is the number of edges of the graph. Each vertex is inspected once directly, and once for each of its in-edges.

Every acyclic graph has a topological sort, which may be found by depth-first search. Consider the sequence of calls to 'Postorder'. By induction, it is easy to prove that a vertex can only appear in the call sequence after all its successors. Therefore, in the *reverse* of the call sequence, it will appear *before* all its successors. This is the defining property of a topological sort.

If a graph is cyclic, the cycles are easily detected during the depth-first search. It is merely necessary to keep track of the set of vertices with active calls of 'DFS' (those that have been visited in preorder, but not yet visited in postorder). If an attempt is made to call 'DFS' for an already active vertex, it must be its own successor in the graph, and therefore a member of a cycle.

Given a cyclic graph, there is an efficient algorithm due to Tarjan that finds the strong components of a graph in a single depth-first search [Tarjan 1972]. Furthermore, the strong components are isolated in reverse topological order.

During a depth-first search of the graph, each vertex is assigned a 'preorder visit number', that is, the vertices are numbered in the order they are first visited. A second number is also associated with each vertex, which we call the 'marking' of the vertex. Initially, each vertex is unmarked (or marked with zero), to indicate that it has not yet been visited. When a vertex is first visited, it is re-marked with the same value as its preorder visit number.

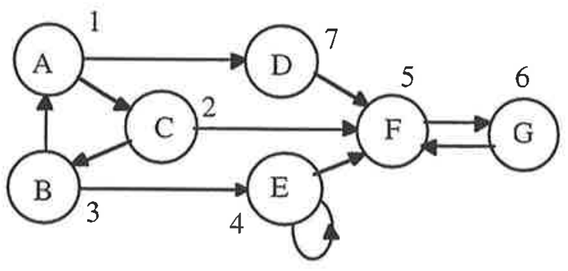

FIGURE 4.8.1: STRONG COMPONENTS OF A GRAPH

Figure 4.8.1 shows the preorder numbering of Figure 4.7.1, assuming that vertices and edges are always chosen in alphabetical order. The order of calls to 'Preorder' is 'A,C,B,E,F,G,D'.

Consider a vertex that happens to be the first vertex that is visited in a strongly-connected component, say 'A' in component {A,B,C}. Before 'A' is finally visited again in postorder, every other vertex of the strongly-connected component will have been visited, i.e., 'B' and 'C', because they among its successors. Each of these vertices must receive a higher preorder visit number than 'A'. (They cannot have been visited before 'A', or 'A' would not be the first vertex visited in the strong component.) Moreover, unless the component is trivial, at least one of its vertices must have a back edge, i.e., an edge connecting it to a vertex with a lower preorder visit number, i.e., 'B' has an edge back to 'A'. (By definition, there must be a path from every vertex of the strongly-connected component to every other, so at least one back edge must enter 'A'.) Whenever an edge is found from a vertex to one with a smaller marking, it is re-marked with the smaller value, i.e., B's marking would be changed from '3' to '1'. As the postorder visits are made, every member of the strongly-connected component — except the first vertex visited — will eventually receive a marking that is less than its own preorder visit number, i.e., C's marking would also be changed to '1'.

This information is used as follows. As each vertex is visited in preorder, it is numbered, marked, and pushed onto a stack. If a vertex still has a marking equal to its preorder visit number when it is visited again in postorder, it was the first vertex visited in a strongly-connected component. In fact, the other vertices of its component lie immediately on top of it on the stack, e.g., 'B' and 'C' lie on top of 'A'. During the postorder visit to 'A', these vertices are popped from the stack and assigned to the current strong component, until the vertex whose marking equals its own preorder numbering ('A' in this case) has been popped and assigned.

Although at one point during the search 'A', 'B' and 'C' are covered on the stack by 'E', 'F' and 'G', and at another point by 'D', they are removed from the stack during the postorder visits to 'F', 'E' and 'D'. This must always happen. Any successors of the vertex in question that belong to other components are removed when visits to those components are completed. This must occur before its post-order visit, because of the acyclic property of any reduced graph.

Once a vertex has been assigned to a strongly-connected component the algorithm marks it with a number greater than the number of vertices, to prevent it causing problems later. This could happen during the visit to 'D', where the edge leading to 'F', which otherwise would have a marking of '5', would be mistaken for a back edge, causing 'D' to marked incorrectly with '5' also.

During a depth-first search, while visiting one strongly-connected component, an edge may be found leading to a successor component. The visit to the successor component must be complete before the visit to the original component is resumed. Thus, the visits to components are completed in a reverse topological sort of the reduced graph. This means that the list of strong components discovered by the algorithm is a reverse topological sort of the vertices of the reduced graph.

Tarjan's algorithm is remarkably efficient. Each vertex and edge of the graph is processed only once.

Another property of a graph that may need to be established is whether a path exists between a given pair of its vertices. This property is given by its transitive closure. The transitive closure of graph G has an edge corresponding to every edge or compound path of G. In the transitive closure, each vertex has an edge to all its descendants in the original graph. Since a depth-first search from a given vertex eventually visits all its descendants, the transitive closure of a graph may be efficiently computed using another variation of Tarjan's algorithm.

Suppose vertex w is a successor of vertex v. Every vertex that is a descendant of w is also a descendant of v. During a depth-first search of an acyclic graph, the postorder visit to w must be completed before the postorder visit to v. Assuming that the descendants of w have been found at the completion of the postorder visit to w, on resuming the visit to v, v can simply inherit w and its descendants as its own. By the time the postorder visit to v itself is completed, it will have inherited the descendants of all its successors.

When the graph is cyclic, the situation is more complex because although w is a descendant of v, v may also be a descendant of w. Thus if v is visited before w, a depth first search will not visit v again, so it will not find all the descendants of w. However, the approach still works correctly for the first vertex visited in each strong component. Since every vertex of a strong component has a path to every other vertex, they all share the same set of descendants as the first vertex visited. The Eve and Kurki-Suonio algorithm [Eve & Kurki-Suonio 1997] is similar to Tarjan's strong component algorithm, but in addition, it keeps track of the descendants of vertices. As each strong component is identified, it assigns the set of descendants of its first vertex visited to all the vertices of the component as they are popped from the stack.

Although it finds the transitive closure during one depth-first search of a graph, the time taken by the Eve and Kurki-Suonio algorithm is usually dominated by recording the edges of the resulting closure, which may number as many as the square of the number of vertices.

4.9 A Worked Example

To illustrate all the steps involved in deriving a CPG from a specification, consider the student record system of Example 4.9.1. Three events are specified: 'Admit' acknowledges that the student has satisfied the admission criteria, 'Set_Quota' makes a class available by setting a non-zero quota, and 'Enrol' records the enrolment of a student. The intention of the specification is that a student may enrol in a class only after satisfying some admission criteria, and provided the class is not already full. (Many realistic details have been omitted from the specification. For example, what happens when a class quota is reduced?)

```
generic
   type student is private;
   type class is private;
package Student_Records is
   procedure Admit (s : student);
   procedure Set_Quota (c : class; n : natural);
   procedure Enrol (c : class; s : student);
end Student_Records;

package body Student_Records is
   Admitted : array (student) of boolean := (others => false);
   Enrolled : array (class, student) of boolean := (others => (others => false));
   Quota : array (class) of natural := (others => 0);
   Size : array (class) of natural := (others => 0);
   procedure Admit (s : student) is
   begin
      Admitted(s) := true;
   end Admit;
   procedure Set_Quota (c : class; n : natural) is
   begin
      Quota(c) := n;
   end Set_Quota;
   procedure Enrol (c : class; s : student) is
   begin
      if Admitted(s) then
         if not Enrolled(c, s) then
            if Size(c) < Quota(c) then
               Size(c) := Size(c) + 1;
               Enrolled(c, s) := true;
            end if;
         end if;
      end if;
   end Enrol;
end Student_Records;
```

EXAMPLE 4.9.1: A STUDENT RECORD SYSTEM

The first step in the analysis of the specification is to find its state variable dependences. From the 'Admit' event we find that 'Admitted' depends on 'Admit.s'. From the 'Set_Quota' event we find that 'Quota' depends on 'Set_Quota.c' and 'Set_Quota.n'. The 'Enrol' event is more complex:

1. From the use of the expression in the outermost **if** statement we get 'Admitted' depends on 'Enrol.s'. (This is an application of Rule 5 of Section 4.5.)

2. From the expression in the second **if** statement we get ‘Enrolled’ depends on ‘Enrol.c’, ‘Enrol.s’, and ‘Admitted’. (A more complex application of Rule 5: the dependence on ‘Admitted’ ensures that if the second **if** statement is encapsulated in a delayed procedure, the call will be nested within the outer **if** statement.)

3. From the expression in the innermost **if** statement, we determine that both ‘Size’ and ‘Quota’ depend on ‘Enrol.c’, ‘Enrol.s’, ‘Admitted’ and ‘Enrolled’. (Rule 5 again.)

4. From the first assignment we get ‘Size’ depends on itself, ‘Enrol.c’, ‘Quota’, ‘Enrolled’, ‘Enrol.s’, and ‘Admitted’. (Rules 1, 2, and 3.)

5. From the second assignment we get ‘Enrolled’ depends on itself, ‘Enrol.c’, ‘Quota’, ‘Size’, ‘Enrol.s’, and ‘Admitted’. (Rules 1, 2, and 3 again.)

The SDG of this specification is more complex than any previous example. Its full graph would look too cluttered. Figure 4.9.1 adopts the convention of showing only its transitive reduction. Since dependence relations are pseudo-transitive, this simplification will not affect the resulting CPG.

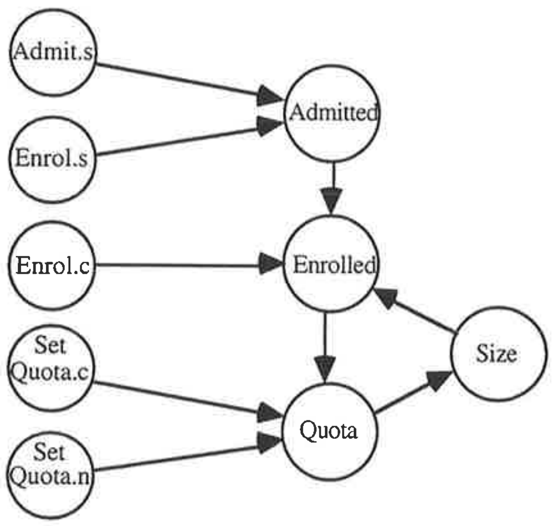

FIGURE 4.9.1: SDG FOR THE STUDENT RECORD SYSTEM

Figure 4.9.1 has a non-trivial strong component containing ‘Enrolled’, ‘Size’, and ‘Quota’. Figure 4.9.2 shows its CPG. The graph shows five input processes, which represent the sources of the parameters of the three events that the system handles. These source processes lie outside the system being specified. Usually, unless we wish to discuss the environment of a system, little is lost by ignoring input processes. In this example the system proper contains two minimal separable processes. It may be implemented by the pipeline of Figure 4.9.3.

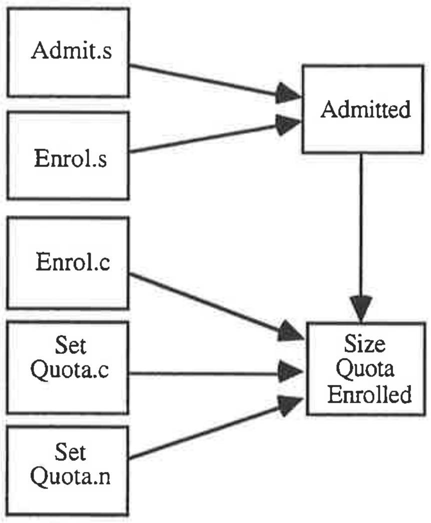

FIGURE 4.9.2: CANONICAL PROCESS GRAPH FOR THE STUDENT RECORD SYSTEM

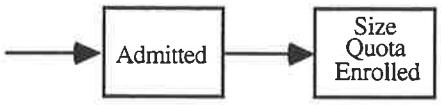

FIGURE 4.9.3: PROCESS PIPELINE FOR THE STUDENT RECORD SYSTEM

Neglecting parallelism or sequential access, there are exactly two possible implementations: join the minimal processes by a queue, or compose the two processes into a single process whose specification is exactly that of Example 4.9.1. However, independent access cannot be neglected. Although it cannot be deduced from the SDG, it turns out that there can be a separate copy of the '{Admitted}' process for each 'student', and a separate copy of the '{Size, Quota, Enrolled}' process for each 'class'. (Considering 'Enrolled' as a matrix, each of its rows may be accessed concurrently, but not each of its elements.) The parallel implementation may be represented by the process graph of Figure 4.9.4, which shows the processes for two possible students and two possible classes. How independence can be detected is addressed in the next chapter.

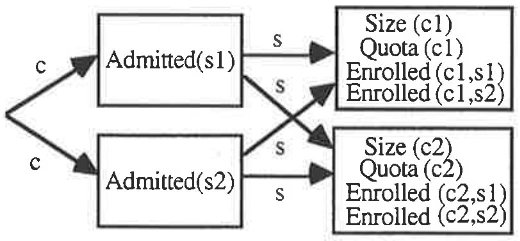

FIGURE 4.9.4: PARALLEL PROCESSES FOR THE STUDENT RECORD SYSTEM

The opportunity for parallelism or sequential access is a strong reason to prefer to keep the processes separate. Combining the '{Admitted}' and '{Size, Quota, Enrolled}' processes to yield a single composite process would destroy the opportunity for independence. The composite process could not be made independent by 'student' and still preserve correct 'class' histories. For example, if there is a shortage of places in a class, it could not be guaranteed that the first students to enrol will get the places. The composite process could not be parallel by 'class' and preserve the correct 'student' histories. For example, an 'Enrol' event could overtake an earlier 'Admit' event for the same 'student', causing the enrolment to be rejected. On the other hand, by sorting the calls from each student process, the queuing network that links the two sets of processes in Figure 4.9.4 can ensure that the correct sequence of events enters each class process, exactly as in the sequential and parallel implementations described in Chapter 3.

Chapter 5 discusses an informal method for discovering opportunities for parallelism or sequential access and the related problem of choosing the optimum process composition. To complete this chapter, we derive the component specifications of the Student Record system by using the rewriting rules.

The implementation should contain two internal packages; the first, which accesses 'Admitted', calling delayed procedures in the second, which accesses 'Size', 'Quota' and 'Enrolled'. Where should the procedure calls be placed? There is no problem with the 'Admit' event, which is confined to the first process. 'Set_Quota' is confined to the second process. In 'Enrol' the called procedure must enclose all references to 'Size', 'Quota' and 'Enrolled'. The calling procedure should also include all assignments to 'Admitted' — except there aren't any. Example 4.9.2 shows one way to divide the text. The other is to place the whole procedure in the second process, passing 'Admitted' as a parameter. The question of where best to place delayed procedure calls will be discussed in Chapter 7.

```
procedure Enrol (c : class; s : student) is
begin
   if Admitted(s) then
      ┌───────────────────────────────┐
      │ if not Enrolled(c, s) then    │
      │    if Size(c) < Quota(c) then │
      │       Size(c) := Size(c) + 1; │
      │       Enrolled(c, s) := true; │
      │    end if;                    │
      │ end if;                       │
      └───────────────────────────────┘
   end if;
end Enrol;
```

EXAMPLE 4.9.2: PROCESS BOUNDARY FOR 'ENROL' EVENTS

Choosing the boundary of Example 4.9.2, the resulting component specifications are given in Example 4.9.3. Only the package body is shown. Since all three arrays in the package 'Process_SQE' (which represents the process accessing 'Size', 'Quota' and 'Enrolled') have a first subscript which is a 'class', and because all three of its procedures have a 'class' as a parameter, it is possible to replace 'Process_SQE' by an array of processes, as in Example 4.9.4.

```
package body Student_Records is
   package Process_A is
      procedure Admit (s : student);
      procedure Set_Quota (c : class; n : natural);
      procedure Enrol (c : class; s : student);
   end Process_A;
   package Process_SQE is
      procedure Set_Quota (c : class; n : natural);
      procedure Enrol (c : class; s : student);
   end Process_SQE;
   package body Process_A is
      Admitted : array (student) of boolean := (others => false);
      procedure Admit (s : student) is
      begin
         Admitted(s) := true;
      end Admit;
      procedure Set_Quota (c : class; n : natural) is
      begin
         Process_SQE.Set_Quota (c, n);
      end Set_Quota;
      procedure Enrol (c : class; s : student) is
      begin
         if Admitted(s) then
            Process_SQE.Enrol (c, s);
         end if;
      end Enrol;
   end Process_A;
   package body Process_SQE is
      Enrolled : array (class, student) of boolean := (others => (others => false));
      Quota : array (class) of natural := (others => 0);
      Size : array (class) of natural := (others => 0);
      procedure Set_Quota (c : class; n : natural) is
      begin
         Quota(c) := n;
      end Set_Quota;
      procedure Enrol (c : class; s : student) is
      begin
         if not Enrolled(c, s) then
            if Size(c) < Quota(c) then
               Size(c) := Size(c) + 1;
               Enrolled(c, s) := true;
            end if;
         end if;
      end Enrol;
   end Process_SQE;
   procedure Admit (s : student) is
   begin Process_SQE.Admit (s); end Admit;
   procedure Set_Quota (c : class; n : natural) is
   begin Process_A.Set_Quota (c, n); end Set_Quota;
   procedure Enrol (c : class; s : student) is
   begin Process_A.Enrol (c, s); end Enrol;
end Student_Records;
```

EXAMPLE 4.9.3: COMPONENT PROCESS OF THE STUDENT RECORD SYSTEM

```
package body Process_SQE is
      Enrolled : array (student) of boolean :=  (others => false);
      Quota : natural :=  0;
      Size :  natural := 0;
      procedure Set_Quota (n : natural) is
      begin
         Quota := n;
      end Set_Quota;
      procedure Enrol (s : student) is
      begin
         if not Enrolled(s) then
            if Size < Quota then
               Size := Size + 1;
               Enrolled(s) := true;
            end  if;
         end  if;
      end Enrol;
   end Process_SQE;
```

EXAMPLE 4.9.4: PARALLEL IMPLEMENTATION OF PROCESS_SQE

An independent access form of 'Process_A' may be constructed along similar lines, as shown in Example 4.9.5. There are two nuances. First, the 'Set_Quota' event is not associated with a particular student. To even the load on each student process in a parallel implementation, 'Set_Quota' events may be sent to student processes at random, perhaps by hashing their parameters. (An alternative would be for these events to by-pass 'Process_A', the enclosing package calling an instance of 'Process_SQE' directly; but this raises the problem of it merging two streams of delayed calls.) Second, the parameter 's' has to be passed to the 'Enrol' procedure for student 's', so that it may be forwarded to the 'Enrol' procedure in 'Process_SQE'.

```
package body Process_A is
   Admitted : boolean := false;
   procedure Admit is
   begin
      Admitted := true;
   end Admit;
   procedure Set_Quota (c : class; n : natural) is
   begin
      Process_SQE(c).Set_Quota (n);
   end Set_Quota;
   procedure Enrol (s: student; c : class) is
   begin
      if Admitted then
         Process_SQE(c).Enrol (s);
      end  if;
   end Enrol;
end Process_A;
```

EXAMPLE 4.9.5: PARALLEL IMPLEMENTATION OF PROCESS_A

When a package is decomposed into an array of independent instances, each instance may only refer to array elements (or, as in the case of 'Enrolled', sub-arrays) that share its own index. What makes the transformation possible is that no package instance has a need to refer to an element declared within a different instance of the same package.

5. Independence

Close coupling corresponds to a cycle in the SDG. Separability corresponds to a one-way path. Independence corresponds to the absence of a path. Independence is the property that makes parallel or sequential access possible.

This chapter develops the theory of independence semi-formally — in the sense that it can used rigorously by a human system designer, but is not yet formal enough for a computer algorithm. (This will be remedied in Chapter 8.) It extends the SDG notation to show where independence exists. It shows how independence information can be extracted from the text of a specification. It also makes the distinction between the pure notion of independence, meaning that *some* form of parallel or sequential access is possible, and a pragmatic notion, which determines whether the parallel or sequential update algorithms of Chapter 3 can be used to achieve it.

There are two main sources of independence. First, a single event may inspect or update many elements of an array in an **all** loop independently. Second, *different* events that inspect or update disjoint sets of array elements may proceed independently of one another. A third, minor source of independence can occur when an event inspects or updates a few selected elements of an array.

The rules for constructing SDG's were determined by the capabilities of delayed procedure call; likewise, the rules for independence must be determined by what the sequential or parallel update algorithms of Chapter 3 can do. In the same way that the rewriting rules for delayed procedure call determined dependence, so the transformation into sequential or parallel access must determine independence. Without constraining the rules in this way, analysis might reveal that the elements of an array could be updated independently, but not suggest *how*. Such pathological cases rarely arise in practice, and will be discussed in Chapter 8.

There is a considerable body of work on detecting potential parallelism in loops, as a feature of optimising and parallelising compilers, to which [Chandy & Misra 1988] provides a tutorial introduction, and of which [Bacon *et al.* 1994] gives a comprehensive survey and bibliography. There is sometimes a close connection between this work and the transformations used in this thesis. For example, the basic form of the sequential access algorithm of Section 3.3 can be derived by loop reordering [Bacon *et al.* 1994, Section 6.2], and the parallel access algorithm may be derived from it by an additional data distribution.

Disappointingly, little of this work is relevant to this thesis. For example, one of its chief concerns is to detect loop-carried dependences in cases such as:

```
for i in 1..100 loop
   C(i) := B(i);
   B(i+1) := A(i);
end loop;
```

where ‘C’ depends on ‘A’ through the assignment to ‘B’. In this thesis, an index such as ‘i+1’ is virtually meaningless because ‘i’ would typically represent an object identifier such as an account number or a product code, and there would typically be no logical relationship between the objects identified by ‘i’ and ‘i+1’. On the other hand, this thesis is concerned with analysing loops of the form:

```
for c in customer loop
   r := Sales_Rep (c);
   Commission (r) := Commission (r) + Sales (c);
end loop;
```

in which there appears to be little opportunity for parallelism, because there is read/write contention for the ‘Commission’ array. The potential parallelism can only be exploited after *separating* the ‘Sales_Rep’ and ‘Commission’ processes, an option involving a series of several transformations by an optimising compiler, and not likely to be discovered.

For an optimising compiler to deal with events that don’t have **for** loops, the event read loop would have to be made explicit. This could be captured by treating a batch of events as an array indexed by time. However, to capture the idea of *sorting* events, the array would need to be multi-dimensional, indexed both by time and by master file keys. Sorting would correspond to transposition of this array, or equivalently, to dynamically altering its mapping onto storage. [Bacon *et al.* 1994, Section 7.1] discusses the optimal mapping of arrays, a problem that is NP-complete even in the static case. [Aguilar 1996] also suggests some novel approaches for mapping tasks to processors based on neural nets, genetic algorithms and simulated annealing. In short, although it would be possible to express a batch system in a way that *could* be optimised using established compiler technology, it would be unreasonable to expect any compiler to find the necessary sequence of steps.

Equally disappointingly, the existing work on parallel databases is not useful here either. [Weikum 1995] gives a useful recent review. The work either concerns scientific array-based problems, e.g., [Thakur *et al.* 1996], or database management systems, which focus on satisfying arbitrary queries. Essentially, a database management system is a server for one or more unpredictable clients, e.g., [Kwong & Majumdar 1996]. Since the precedence relations within update transactions are invisible to the server, the onus is on the clients to issue sub-transactions that can be executed in parallel, e.g., [DeVirmis & Ulusoy 1996]. The database management system must resort to data placement strategies based on static relationships between data objects, e.g., [Taniar 1998] and on dynamic load balancing between processors, e.g., [Taniar & Yang 1998]. In this thesis, we assume on the contrary that all possible queries and updates are known at the start of the design process.

Before discussing how independence should be analysed in the context of the thesis, the next two sections discuss some further properties of independent access algorithms, which were not explored in Chapter 3.

5.1 The Treatment of Nested Loops

Chapter 2 illustrated the use of loops in the specification of the 'Audit' event. A similar event to audit the number of enrolments in each class of the Student Records system, shown in Example 5.1.1, is a second example of a specification with a loop. Its aim is to check that the number of enrolments in each row of the 'Enrolled' array equals the corresponding value of 'Size'. If it does not, a 'Bad_Size' event is sent to an externally defined 'Error_Report' system, which can be assumed to display a suitable message to the user.

```
with Error_Report;
generic
   type student is private;
   type class is private;
package Student_Records is
   procedure Audit;
end Student_Records;
package body Student_Records is
   Enrolled : array (class, student) of boolean := (others => (others => false));
   Size : array (class) of natural := (others => 0);
   procedure Audit is
      Count : array (class) of natural := (others => 0);
   begin
      all c in class loop
         all s in student loop
            if Enrolled(c, s) then
               Count(c) := Count(c) + 1;
            end if;
         end loop;
         if Count(c) /= Size(c) then
            Error_Report.Bad_Size (c);
         end if;
      end loop;
   end Audit;
end Student_Records;
```

EXAMPLE 5.1.1: CHECKING THE INTEGRITY OF CLASS SIZES

The specification contains two nested **all** loops. The outer loop can be executed in parallel by independent processes, but the inner loop shares the use of 'Count(c)'. As explained in Section 2.6.4, the assignment to 'Count(c)' is a convention for a reduction operation.

(There is a separate element of 'Count' for each class. The convention for reduction given in Section 2.6.4 requires 'Count' to be initialised in its declaration. With *sequential* execution of the outer loop of Example 5.1.1, it would be possible for 'Count' to be a simple variable, provided it is re-initialised on each iteration. However, the **all** loop allows its iterations to proceed in parallel, so that its initialisations, increments, and inspections would occur unpredictably.)

Example 2.5.3a–b showed an example of modelling independent processes in a library system. Where there are nested loops, as in the example of Example 5.1.1, they may be modelled by nested levels of enclosing packages, or shells within shells. Example 5.1.2 shows the resulting model. The outer shell represents the whole system. It encloses an instance of 'Student_Records_Class' for every value of 'class'. An 'Audit' event sent to the outer shell causes an 'Audit' event to be sent to each instance of 'Student_Records_Class', in parallel. In turn, the 'Audit' procedure in each 'Student_Records_Class' instance invokes an instance of the 'Audit' procedure in 'Student_Records_Class_Student' for each value of 'student'. There is a separate instance of 'Count' for each 'class'. The abstraction is similar to the treatment of the library 'Audit' event in Example 2.5.3a–b.

```
package body Student_Records is
   array (class) of package Student_Records_Class is
      procedure Audit;
   end Student_Records_Class;
   package body Student_Records_Class is
      Size : natural := 0;
      array (student) of package Student_Records_Class_Student is
         procedure Audit (Count : in out natural);
      end Student_Records_Class_Student;
      package body Student_Records_Class_Student is
         Enrolled : boolean := false;
         procedure Audit (Count : in out natural) is
         begin
            if Enrolled then
               Count := Count + 1;
            end if;
         end Audit;
      end Student_Records_Class_Student;
      procedure Audit (c: class) is
         Count : natural := 0;
      begin
         all s in student loop
            Student_Records_Class_Student(s).Audit(Count);
         end loop;
         if Count /= Size then
            Error_Report.Bad_Size(c);
         end if;
      end Audit;
   end Student_Records_Class;
   procedure Audit is
   begin
      all c in class loop
         Student_Records_Class(c).Audit(c);
      end loop;
   end Audit;
end Student_Records;
```

EXAMPLE 5.1.2: PROCESS SPECIFICATIONS FOR CHECKING CLASS SIZES

5.2 Updating Multiple Elements

Consider the simple accounting system of Example 5.2.1, whose 'Transfer' procedure models an event where some money ('Amount') is moved between the accounts 'From' and 'To'.

```
generic
   type account is private;
   type money is range <>;
package Accounting is
   procedure Transfer (From, To: account; Amount : money);
end Accounting;
package body Accounting is
   Balance : array (account) of money := (others => 0);
   procedure Transfer (From, To: account; Amount : money) is
   begin
      Balance(From) := Balance(From) – Amount;
      Balance(To) := Balance(To) + Amount;
   end Transfer;
end Accounting;
```

EXAMPLE 5.2.1: A TRANSACTION BETWEEN TWO ACCOUNTS

```
package body Accounting is
   array (account) of package Accounting_Account is
      procedure Transfer_1 (Amount : money);
      procedure Transfer_2 (Amount : money);
   end Accounting_Account;
   package body Accounting_Account is
      Balance : money := 0;
      procedure Transfer_1 (Amount : money) is
      begin
         Balance := Balance – amount;
      end Transfer_1;
      procedure Transfer_2 (Amount : money) is
      begin
         Balance := Balance + Amount;
      end Transfer_2;
   end Accounting_Account;
   procedure Transfer (From, To: account; Amount : money) is
   begin
      Accounting_Account(From).Transfer_1(Amount);
      Accounting_Account(To).Transfer_2(Amount);
   end Transfer;
end Accounting;
```

EXAMPLE 5.2.2: COMPONENTS OF A TRANSACTION BETWEEN TWO ACCOUNTS

In this example, the two accounts are updated independently, and it does not matter in what order the assignments are done. Provided the value of 'Amount' is transmitted to both the 'To' and the 'From' processes, parallel or sequential updating is possible, as shown in Example 5.2.2. The distributor would need to pre-process each event record to create two calls: one for each account involved. In the unlikely case that 'From' and 'To' are the same account, the atomic natures of 'Transfer_1' and Transfer_2' ensure they are executed serially, so that

'Transfer' causes no change in 'Balance', which is correct. Further, provided the delayed procedure calls are time-stamped properly, they must also be executed in the specified order.

The possibility of the distributor making multiple calls was not discussed in Chapter 3. It might even make an unpredictable number of calls. The example of Example 5.2.1 may be extended to a more general case where a list of accounts must be updated, as when posting a complex journal entry. The distributor process would then have to generate a call for each account in the list. Although several accounts are involved, their processing remains independent.

```
generic
   type account is private;
   type money is range <>;
package Accounting is
   procedure Safe_Transfer (From, To: account; Amount : money);
end Accounting;
package body Accounting is
   Balance : array (account) of money := (others => 0);
   procedure Safe_Transfer (From, To: account; Amount : money) is
   begin
      if Balance(From) >= Amount then
         Balance(From) := Balance(From) – Amount;
         Balance(To) := Balance(To) + Amount;
      end if;
   end Safe_Transfer ;
end Accounting;
```

EXAMPLE 5.2.3: A 'SAFE' TRANSACTION BETWEEN TWO ACCOUNTS

Now consider a version of Example 5.2.1 when a money transfer is valid only if the 'From' account would not become overdrawn, as specified in Example 5.2.3. Such a specification has no parallel implementation because the value of 'Balance(From)' must be inspected before 'Balance(To)' can be updated. In other words, 'Balance(To)' now depends on Balance(From)'. The various 'Balance' processes are no longer independent, and cannot lag one another. Although for a particular transfer, 'Balance(To)' can lag 'Balance(From)', a second transfer might reverse the roles of the two accounts. For example, the first transfer might be from 'Smith' to 'Jones', but the second might from 'Jones' to 'Smith'. No partial ordering can be exploited, because transfers can occur between any pair of accounts in either direction.

Example 5.2.4 shows the effect of trying to make a parallel implementation of Example 5.2.3. It needs to call two procedures, 'Find_Balance' and 'Debit', in the 'From' component. It does not fit the independent update model of Section 3.1 because it has two distributor and collector stages. Two stages cannot be allowed by independent update algorithms, because it would be possible for some other 'Debit' event to change the value of 'Balance(From)' between the two stages. This might decrease the value of 'Balance' to a point where the second stage would cause the 'From' account to become overdrawn. The only way to ensure safety is to make the shell procedure appear atomic by using a locking protocol. This in turn could lead to deadlock — for example, if two concurrent events involved the same two accounts with reversed roles.

This would be moving a long way from the independent access model of Section 3.1. The constraints on an update shell are rigid: the generation of a set of remote procedure calls to one or more element updates, sometimes followed by collecting the returned values. Since the properties of the update algorithms define independence, we have to say that the different accounts are not independent.

```
generic
   type account is private;
   type money is range <>;
package Accounting is
   procedure Safe_Transfer (From, To: account; Amount : money);
end Accounting;
package body Accounting is
   array (account) of package Accounting_Account is
      procedure Find_Balance (Balance : out money);
      procedure Debit (Amount : money);
      procedure Credit (Amount : money);
   end Accounting_Account;
   package body Accounting_Account is
      Balance : money := 0;
      procedure Find_Balance (Bal : out money) is
      begin
         Bal := Balance;
      end Find_Balance;
      procedure Debit (Amount : money) is
      begin
         Balance := Balance – Amount;
      end Debit;
      procedure Credit (Amount : money) is
      begin
         Balance := Balance + Amount;
      end Credit;
   end Accounting_Account;
   procedure Safe_Transfer (From, To: account; Amount : money) is
      Balance : money := 0;
   begin
      Accounting_Account(From).Find_Balance (Balance);
      if Balance >= Amount then
         Accounting_Account(From).Debit(Amount);
         Accounting_Account(To).Credit(Amount);
      end if;
   end Safe_Transfer ;
end Accounting;
```

EXAMPLE 5.2.4: AN ATTEMPT TO PARALLELISE A 'SAFE' TRANSACTION

Does the example of Example 5.2.4 mean that any form of dependence between two accounts prevents parallelism? Certainly not. Example 5.2.5 shows a counter-example. In this modified form of the problem, instead of requiring that the 'From' account does not become overdrawn, we merely require that both accounts are 'Authorised'. In this case, not only does the updating of the 'To' account depend on the 'From' account, but the updating of the 'From' account depends on the 'To' account.

```
generic
   type account is private;
   type money is range <>;
package Accounting is
   procedure Careful_Transfer (From, To: account; Amount : money);
end Accounting;
package body Accounting is
   Authorised : array (account) of boolean := (others => false);
   Balance : array (account) of money := (others => 0);
   procedure Careful_Transfer (From, To: account; Amount : money) is
   begin
      if Authorised(From) and Authorised(To) then
         Balance(From) := Balance(From) – Amount;
         Balance(To) := Balance(To) + Amount;
      end if;
   end Careful_Transfer;
end Accounting;
```

EXAMPLE 5.2.5: A 'CAREFUL' TRANSACTION BETWEEN TWO ACCOUNTS

```
package body Accounting is
   package Accounting_A is
      procedure Careful_Transfer (From, To: account; Amount : money);
   end Accounting_A;
   package body Accounting_A is
      array (account) of package Accounting_A_Account is
         procedure Find_Authorised (Auth: out boolean);
      end Accounting_A_Account;
      package body Accounting_A_Account is
         Authorised : boolean := false;
         procedure Find_Authorised (Auth: out boolean) is
         begin
            Auth := Authorised;
         end Find_Authorised;
      end Accounting_A_Account;
      procedure Careful_Transfer (From, To: account; amount : money) is
         Authorised_To, Authorised_From : boolean;
      begin
         Accounting_A_Account(To).Find_Authorised(Authorised_To);
         Accounting_A_Account(From).Find_Authorised(Authorised_From);
         if Authorised_To and Authorised_From then
            Accounting_B.Careful_Transfer_From(From);
            Accounting_B.Careful_Transfer_To(To);
         end if;
      end Careful_Transfer;
   end Accounting_A;
```

EXAMPLE 5.2.6A: THE 1ST COMPONENT OF A 'CAREFUL' TRANSFER

Despite this double interaction, Example 5.2.6a and Example 5.2.6b show how the specification of Example 5.2.5 can be implemented by two parallel update processes. The first update accesses the values of 'Authorised' and the second accesses the values of 'Balance'. The two processes are separable. The factorisation into two separable processes is analogous to the library system example of Example 2.5.3, except that, whereas in that case the two processes have different domains ('users' and 'books'), in this case they have the same domain.

```
package Accounting_B is
   procedure Careful_Transfer_To (To: account; Amount : money);
   procedure Careful_Transfer_From (From: account; Amount : money);
end Accounting_B;
package body Accounting_B is
   array (account) of package Accounting_B_Account is
      procedure Careful_Transfer_To (Amount : money);
      procedure Careful_Transfer_From (Amount : money);
   end Accounting_B_Account;
   package body Accounting_B_Account is
      Balance : money := 0;
      procedure Careful_Transfer_To (Amount : money) is
      begin
         Balance := Balance + Amount;
      end Careful_Transfer_To;
      procedure Careful_Transfer_From (Amount : money) is
      begin
         Balance := Balance – Amount;
      end Careful_Transfer_From;
   end Accounting_B_Account;
   Balance : array (account) of money := (others => 0);
   procedure Careful_Transfer_To (To: account; amount : money) is
   begin
      Accounting_B_Account(To).Careful_Transfer_To (amount);
   end Careful_Transfer_To;
   procedure Careful_Transfer_From (From: account; Amount : money) is
   begin
      Accounting_B_Account(From).Careful_Transfer_From (Amount);
   end Careful_Transfer_From;
end Accounting_B;
procedure Careful_Transfer (From, To: account; Amount : money) is
begin
   Accounting_A.Careful_Transfer (From, To, Amount);
end Careful_Transfer;
end Accounting;
```

EXAMPLE 5.2.6B: THE 2ND COMPONENT OF A 'CAREFUL' TRANSFER

These examples show how pre-processing and post-processing in the update shell can increase the power of a set of independent processes, either by directing messages sent to them, by coordinating their access to a shared variable, or by containing some part of an event procedure that requires access to more than one element. The existence of the shell also means that a process does not need to know whether a second process to which it sends an output uses independent access. The output may always be safely sent to its enclosing shell.

5.3 Showing Independence on State Dependence Graphs

Although SDG's have proved adequate for associating processes with sets of attributes, they have not yet thrown any light on the question of independence. This is because their vertices have represented whole arrays, rather than single elements. The solution is draw a vertex for each element. Unfortunately, realistic problems involve arrays with many elements, so the resulting graphs would be too large for practical use. But by pretending that domains have very few elements (e.g., 2 or 3), it becomes possible to draw and understand SDG's that have a vertex for every element. Figure 5.3.1 models the 'Audit' event of Example 5.1.1, in the case that there are two classes and two students. The vertices 's(1)' and 's(2)' represent the two different definitions of 's' that occur in the two iterations of the body of its outer loop. (This is a case where it is impossible to ignore the distinction between a variable and its definitions. Each iteration of the outer loop body involves a separate definition of 's'.)

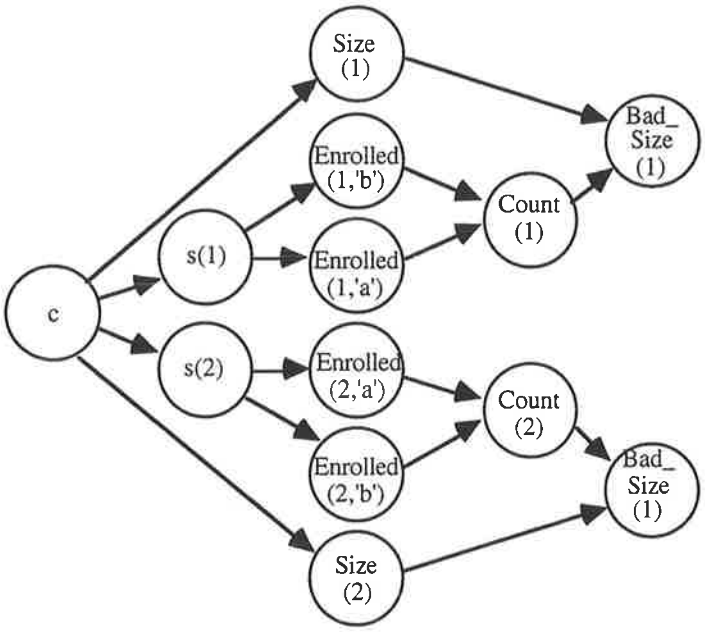

FIGURE 5.3.1: MODELLING PARALLELISM

Figure 5.3.1 does not show what might be expected. It suggests that 'Enrolled' and 'Count' could be allocated to separate processes, whereas the implementation modelled in Example 5.1.2 combines them into a single process. Indeed, it is possible to separate them, with the 'Enrolled' process making delayed calls to a procedure in the 'Count' process that contains the simple assignment 'Count(c):=Count(c)+1'. The truth is, that is exactly how the parallel implementation of Section 3.5 works: partial sums are forwarded to a collector process; the 'Count' process *is* the collector process.

It is interesting to contrast the SDG's shown in Figure 5.3.2, Figure 5.3.3, and Figure 5.3.4 for the three different transactions between accounts that were specified in Example 5.2.1, Example 5.2.3, and Example 5.2.4. Note especially the dependences between the elements of 'Balance'.

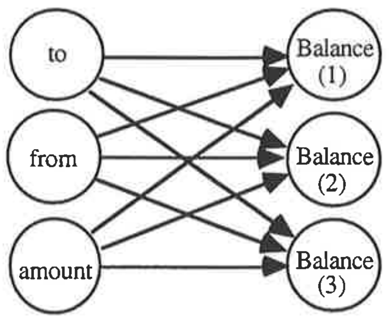

FIGURE 5.3.2: THE 'TRANSFER' SDG

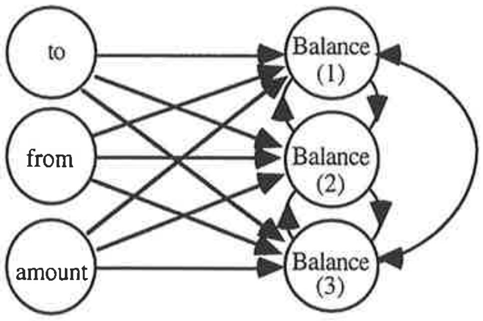

FIGURE 5.3.3: THE 'SAFE TRANSFER' SDG

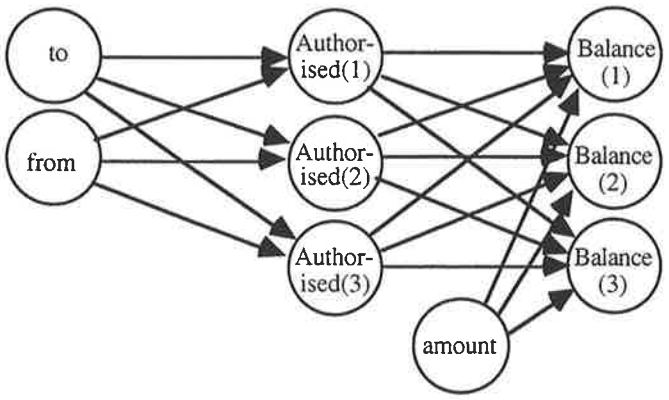

FIGURE 5.3.4: THE 'CAREFUL TRANSFER' SDG

In Figure 5.3.2, each 'Balance' element is independent of the others. In Figure 5.3.3, each 'Balance' depends on all the others. (This arises because the value of 'Balance(To)' depends on 'Balance(From)', and we assume that 'To' and 'From' can have any of the values 1, 2, or 3.) In Figure 5.3.4, although there are dependences between accounts — because for example, 'Balance(From)' depends on 'Authorised(To)' — there are no dependences between the elements of 'Balance' or between the elements of 'Authorised'. Figure 5.3.2 allows parallel or sequential access because the balances are independent. Figure 5.3.3 does not, because the balances are strongly connected. Figure 5.3.4 allows parallel access provided that the 'Authorised' and 'Balance' arrays are accessed in separate processes. In this example, the shells enclosing the parallel 'Authorised' and 'Balance' processes allow the delayed calls to be switched between different accounts. Without this intervening switching network, the parallelism would be impossible, exactly as in the case of the library system of Figure 1.4.1 (Page 6).

Analysing independence is therefore a simple extension of earlier principles. In Figure 5.3.3, the elements of 'Balance' are strongly connected, so they form one separable process. In Figure 5.3.2 and Figure 5.3.4 they are unconnected, so they may be processed independently.

Suppose, in a modification of the specification of Example 5.2.4, the values of 'Authorised' also depended on 'Balance', perhaps because of the requirements of some new event, e.g., a periodic check to 'de-authorise' accounts whose balances have exceeded their budget. Then the SDG for a system that could implement both requirements would contain edges connecting each element 'Balance' to the corresponding element of 'Authorised', as in Figure 5.3.5. These edges would create cycles, the 'Balance' and 'Authorised' processes would no longer be separable. But worse still, because of the cross connections between accounts made by the edges from 'Authorised' to 'Balance', all the 'Authorised' and 'Balance' elements would be strongly connected; they would belong to one minimal separable process, therefore their independence would be destroyed too.

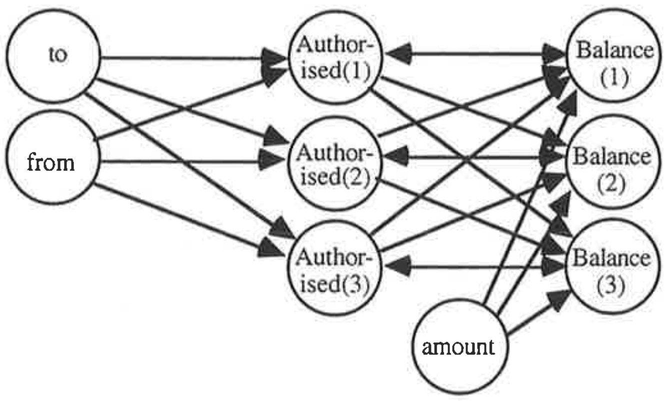

FIGURE 5.3.5: 'CAREFUL TRANSFER' PLUS BUDGET CHECKING

5.4 Compatibility and Conflict

Given the text of an event procedure, how can it be decided when it has independent components? Independence is possible when the elements of a single array in an SDG are unconnected. In general, a process can update several arrays using independent access, even if elements from different arrays are connected, provided that the sets of elements for each index are not connected to one another. This was the case for the 'Size' and 'Quota' arrays in Example 4.9.1, where each 'class' can be processed independently. The subgraphs of the dependence diagram for each 'class' are independent. As in this example, if the elements of two or more arrays allow independent access when they share the same process, the arrays will be said to 'agree' or to be 'compatible'. In terms of an implementation, it will usually pay to store the attributes they model in the same file or table and access them in one hit.

Example 4.9.1 also illustrated *limited* independence involving the 'Enrolled' array. In this case, the 'Enrolled' array could be accessed independently by 'class', but not by 'student'. Compatibility is not all or nothing. The 'Enrolled' array may be said to be 'many-one compatible' with the 'Size' and 'Quota' arrays, or more accurately, 'compatible with respect to class'.

We have seen that the strong components of an SDG determine the separable processes of an implementation. Where two separable processes are connected by a path, the direction of the path determines the direction of data flow, and the order in which the processes may be executed. But when there is no directed path between two processes, they are independent. The two processes could be executed in either order, or concurrently. However, we would not say that the two processes were 'compatible' unless they accessed files or arrays with related domains, because one sequential or parallel update can only deal with hierarchically related domains. We would expect unrelated domains to be updated by separate processes.

The notion of compatibility serves two purposes. First, if several arrays are accessed within the same process, their compatibilities determine its maximum degree of parallel or sequential access. Second, if two processes are compatible with each other, it may pay to combine them into a single 'composite' process. There are two reasons why composing them is almost certain to improve the efficiency of the implementation. First, it may eliminate the overheads of a queue connecting the processes. Second, provided the attributes they access are stored in the same file or table, they may be accessed by only one retrieval. In contrast, combining two incompatible processes would destroy an opportunity for parallel or sequential access.

Partial compatibility is an intermediate case. Combining two partially compatible processes would pay in a parallel implementation — unless the number of elements of their common domain were fewer than the number of physical processors. But it would not usually pay in a sequential implementation, because it would introduce (clustered) random access to at least one table. Therefore, the optimisation of a system for parallel access differs slightly from its optimisation for sequential access.

A simple lexical rule that makes it possible to recognise compatibility between two variables is that they have the same index values in the text of the event procedure; e.g., in the assignment 'A(j):=B(j)', 'A(j)' is compatible with 'B(j)'. Where the variables have different numbers of indices, the common elements determine their degree of compatibility. For example, if 'i' is of type 'T', 'C(i,j,k)' and 'D(i,n)' are partially compatible with respect to T.

It is important to be sure that the values of the indices are the same in each of their uses, e.g., that 'j' has the same value in 'A(j)' as it does in 'B(j)'. For simplicity, variables may be considered to agree only if the indices concerned are constant — e.g., given by input parameters of the procedure, or are defined by an **all** loop. This rule is 'safe'; conflicting variables will never be considered to be compatible, but some pairs of compatible variables may be overlooked.

(It would be more general to test whether indices had equal values, rather than equal names. However, this would mean interpreting the values of expressions, which is not consistent with the variable-oriented approach to dependence discussed in Section 4.5. It would be more general still to detect when elements were accessed in one-to-one correspondence, as in 'A(j) := B(j+1)'. This is an important topic in the optimisation of parallel algorithms [Chandy & Misra 1988], but is not discussed here. Identifiers used in databases are often assigned to objects rather arbitrarily, and rarely have any useful arithmetic properties.)

However, even elements that have different indices may have compatibility of a kind, as illustrated by Example 5.2.2, where 'Balance(To)' and 'Balance(From)' may be accessed in parallel. The essential point is that if an assignment of the form 'Balance(To) := Balance(From)' had been present, it would have destroyed the compatibility. Compatibility is therefore the absence of conflict. Conflict arises because of a dependence between variables having different indices. These cases give the following rules:

1. If two arrays 'A' and 'B' have domains (D,E,F) and (D,E,G,H), 'A' and 'B' are compatible *at most* with respect to (D,E).

2. If variable 'A(i,j,k)' depends on 'B(i,j,m,n)' then 'A' and 'B' are compatible *at most* with respect to the domain of '(i,j)'.

(These rules are easily generalised to any number of indices. In the case of simple variables, which have no indices, two simple variables are compatible, and a simple variable is compatible one-to-many with any indexed variable.)

If there are several compatibilities established by Rules 1 and 2, their smallest domain should be chosen, e.g., two arrays 'A' and 'B' are compatible (maximally) with respect to (D,E), provided they agree at most with respect to (D,E), but are nowhere compatible only with respect to (D) alone.

If array 'A' has domain (D,F) and 'B' has domain (E,F), 'A' and 'B' are considered here to conflict, not be partially compatible with respect to 'F'. This supposes that a physical implementation of 'A' will be primarily clustered according to 'D' and the physical implementation of 'B' will be primarily clustered according to 'E'. To make 'A' and 'B' partially compatible, their domains would have to be specified as '(F,D)' and '(F,E)' instead.

Choosing the best domain hierarchy is part of the specification problem. Thus if the 'Enrolled' array of Example 5.1.1 had been declared with the basis '(student, class)' rather than '(class, student)', it would have destroyed the parallelism by 'class' that was exploited in Example 5.1.2. In the case of sequential access, it is the common prefix that counts, because that corresponds to the sorted order of the master files; in the case of parallel access, it is necessary to decide how they will be mapped to processors. We may assume without loss of generality that the mapping is indicated by the order of the domains, so the same rule applies.

It is important to treat local variables carefully. Because each invocation of an event procedure creates new temporary instances of its local variables, they enjoy a freedom not given to state variables. Consider the sequence of Example 5.4.1.

```
begin
   t := A(i);
   B(i) := t;
end;
```

EXAMPLE 5.4.1: IMPLICIT AGREEMENT

If ‘t’ is a local variable, it is compatible with both ‘A(i)’ and ‘B(i)’. The effect would be exactly the same if ‘t’ was declared as an array, and both occurrences of ‘t’ were replaced by ‘t(i)’, as in Example 5.4.2.

```
begin
   t(i) := A(i);
   B(i) := t(i);
end;
```

EXAMPLE 5.4.2: EXPLICIT AGREEMENT

Compatibility is not transitive. In the sequence of Example 5.4.3, ‘t’ is compatible with ‘A(i)’ or ‘B(j)’ separately, but it cannot be compatible with both at once.

```
begin
   t := A(i);
   B(j) := t;
end;
```

EXAMPLE 5.4.3: AN AGREEMENT CONFLICT

Unfortunately, it is not possible to deal with these last two situations well while using a definition of dependence that is based on names rather than definitions of variables. For the present, the reader may either rely on common sense, or may consider all four assignments in Example 5.4.1 and Example 5.4.3 to conflict. In the case of Example 5.4.1, the specification should be rewritten as in Example 5.4.2. Example 5.4.3 may be rewritten either as Example 5.4.4 or Example 5.4.5. In one case ‘t’ conflicts with ‘A’, and in the other ‘t’ conflicts with ‘B’.

```
begin
   t(j) := A(i);
   B(j) := t(j);
end;
```

EXAMPLE 5.4.4: CONFLICT WITH ‘A’

```
begin
   t(i) := A(i);
   B(j) := t(i);
end;
```

EXAMPLE 5.4.5: CONFLICT WITH ‘B’

With this simplified view of compatibility, like dependence, it becomes a pseudo-transitive relationship. Conversely, Example 5.4.6 shows that *conflict* is intransitive; ‘t’ conflicts both with ‘A’ and with ‘B’, although ‘A’ and ‘B’ agree — they share the same domain, and there is no dependence between them.

```
begin
   A(j) := t(i);
   B(j) := t(i);
end;
```

EXAMPLE 5.4.6: UNDIRECTED INTRANSITIVITY OF CONFLICT

```
begin
   t(i) := A(j);
   B(j) := t(i);
end;
```

EXAMPLE 5.4.7: DIRECTED PSEUDO-TRANSITIVITY OF CONFLICT

However, considered as a directed property based on dependence, even conflict behaves pseudo-transitively. In Example 5.4.7 there is a conflict from ‘A’ to ‘t’, and a second conflict from ‘t’ to ‘B’. In the sense that they could not be accessed by the same set of independent processes, ‘A’ and ‘B’ conflict. Any proposed {A,B} process would be strongly connected to the ‘t’ process, so that the {A,B} and ‘t’ processes could not be separated. The resulting {A,B,t} process would allow no independence, because ‘A’ and ‘B’ conflict with ‘t’.

5.5 Showing Compatibility Information in Graphs

Several examples have shown that it is cumbersome to draw SDG’s even when each domain is limited to contain only 2 or 3 elements. It is more convenient to draw SDG’s whose vertices represent entire arrays, and then label them with compatibility information. Two new conventions make it possible for a system designer to construct labelled SDG’s directly from the texts of event procedures.

The first convention is to colour the vertices according to their domains. If two vertices have the same colour, they have the same domain, and potentially agree. Two vertices with different colours certainly conflict. This scheme has the minor drawback that it does not explicitly show many-one, or partial, agreement. Thus, in interpreting a graph, the viewer may have to remember, for example, that the grey vertices are many-one compatible with the black vertices. In practice, little such information needs to be remembered, and the convention of assigning colours to domains works well.

Apart from those between domains, conflicts also arise from dependences between variables. Even two arrays with the same domain may conflict, as in the assignment ‘A(i) := B(j)’, where ‘i’ and ‘j’ have the same domain. Dependences are represented by edges in the SDG. The second convention is to mark edges with a cross if they cause a conflict, and leave them unmarked if they don’t. However, to prevent visual clutter, edges that join differently coloured vertices are left unmarked. In other words, a cross means that a conflict exists between compatible domains. When two domains have a many-one agreement (e.g., between ‘(Class, Student)' and 'Class'), the same basic rule applies. An edge joining them is marked only if it destroys the many-one agreement. If there are several dependences between two vertices, only one edge is drawn; but if *any* of the dependences conflicts, it must be marked.

(It is possible to find intermediate situations, as in the assignment 'A(i,j) := B(i,k)', where 'j' and 'k' share the same domain. Here, the edge to 'A' from 'B' would need to be marked by a cross, and some additional footnote might need to be added to the graph to make it clear that 'A' and 'B' still agree partially with respect to the domain of 'i'. Such situations are rare enough in practice for this to be a minor notational problem.)

Finally, we must take care when drawing a transitive reduction of an SDG that we do not omit edges that convey essential conflict information. For example, the following assignments could arise separately in three different events; 'A(i) := B(i)', 'B(i) := C(i)', and 'A(i) := C(j)'. Ordinarily, an SDG would not need to show an edge from 'C' to 'A' as there is also a path from 'C' to 'A' via 'B'. However, the edge would need to be drawn to record the conflict between 'A' and 'C'. As a special case, it may even be necessary to draw a loop on a vertex, as in the assignment 'A(i) := A(j)'.

Despite these caveats, the annotated SDG's for the earlier examples are easy to understand. Figure 5.5.1 shows the simplification of Figure 5.3.1. None of its edges are marked because all the dependences agree as far as they can. For example, the edge from 'Enrolled' to 'Count' is many-one, and agrees partially with respect to 'class'.

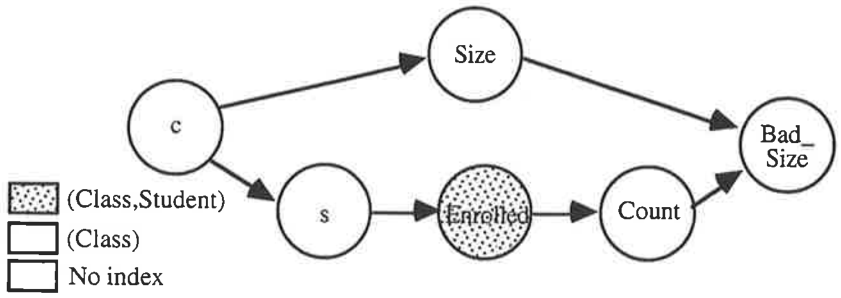

FIGURE 5.5.1: AUDITING CLASS SIZES

Figure 5.5.2 is the simplification of Figure 5.3.2, which describes the unconditional transfer of an amount between two accounts. There are no conflicts due to dependences. This should be contrasted with Figure 5.5.3 (which is the simplification of Figure 5.3.3), showing the SDG of a conditional transfer. The graph includes a loop on 'Balance'. The loop is marked, to show the dependence of 'Balance(To)' on 'Balance(From)', which destroys the opportunity for independent access.

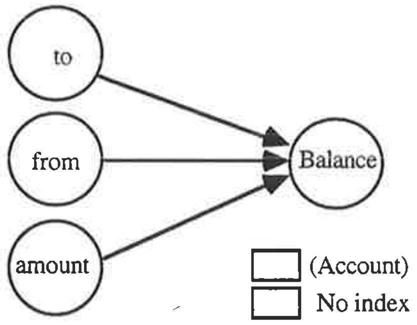

FIGURE 5.5.2: THE 'TRANSFER' SDG

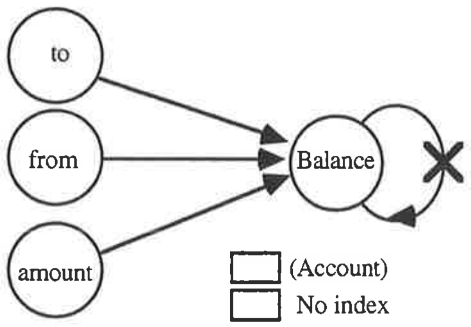

FIGURE 5.5.3: THE 'SAFE TRANSFER' SDG

Figure 5.5.4 is a simplification of Figure 5.3.4, and shows the case where a transfer is possible only between authorised accounts. The edge from 'Authorised' to 'Balance' is marked to show the conflicting dependences. The 'Authorised' and 'Balance' processes are separable, and both processes may use independent access. The conflicting dependences are handled by the queuing network. Recalling Example 5.2.5, the shell enclosing the 'Authorised' process instances is responsible for collating their outputs, and forwarding a single delayed call to the 'Balance' shell. The 'Balance' shell is responsible for splitting this call into two. However, if a single process were to access both 'Authorised' and 'Balance', it would contain the conflicting dependence, and would need to use random access.

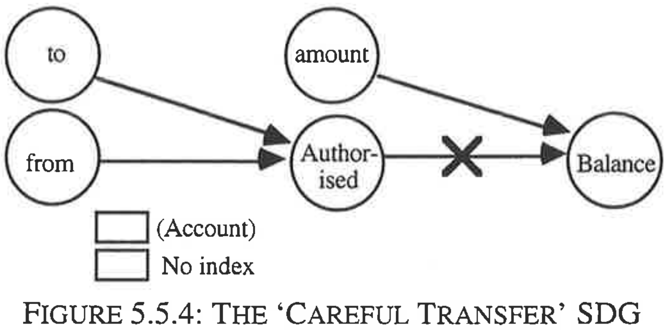

FIGURE 5.5.4: THE 'CAREFUL TRANSFER' SDG

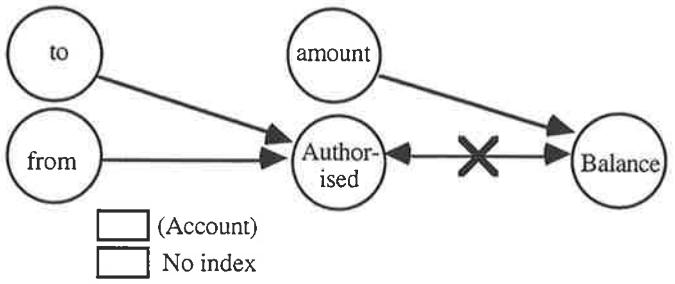

FIGURE 5.5.5: 'CAREFUL TRANSFER' AND BUDGET CHECKING

Figure 5.5.4 must be contrasted with Figure 5.5.5, which shows the effect of adding the budget checking event to the specification. Because of the two-way dependence between 'Authorised' and 'Balance', they must be assigned to the same minimal process. Independence by 'Account' is now impossible, because the minimal process contains a marked edge.

5.6 Recursive Structures

Figure 5.3.3 showed a case where there are no directed paths between the elements of an array (i.e., 'Balance'), and Figure 5.3.4 showed a case where they are totally connected. The total connectivity arose because we assumed that all accounts are equal: a transfer may occur from any account to any other account. Are there any cases intermediate between independence and total connectivity? Possible candidates are graphs whose vertices are not totally connected, and which are either cyclic or acyclic. The cyclic cases may be reduced to acyclic cases by considering their reduced strong component graphs, i.e., the vertices forming each strong component are clustered together, resulting in an acyclic graph of clusters. The resulting clusters represent minimal separable processes. The clusters therefore form some kind of partial ordering, and have at least one topological sort. It should therefore be possible to process the clusters in topological order.

Are there any real-world problems that have this characteristic? At least two: one is the 'Bill of Materials Problem', a second is the 'Chart of Accounts Problem'. They prove to be outside the scope of the Canonical Decomposition Method. Nonetheless, they are worth a brief discussion as a footnote to this chapter.

The Bill of Materials Problem is really a set of problems that involve products assembled from parts. Each 'final assembly' is assembled from several 'sub-assemblies'. Each subassembly is assembled from lower-level sub-assemblies or from 'basic parts'. Basic parts are manufactured or purchased, not assembled. Consider an ordinary pair of spectacles. They are structured as in Figure 5.6.1, where the labels on the edges indicate the numbers of parts required, e.g., there are 4 screws altogether. (The screws that clamp the lenses are assumed to be similar to the screws that hinge the side-frames to the lens mount.) The structure is a Hasse diagram that defines the ways in which the spectacles may be assembled.

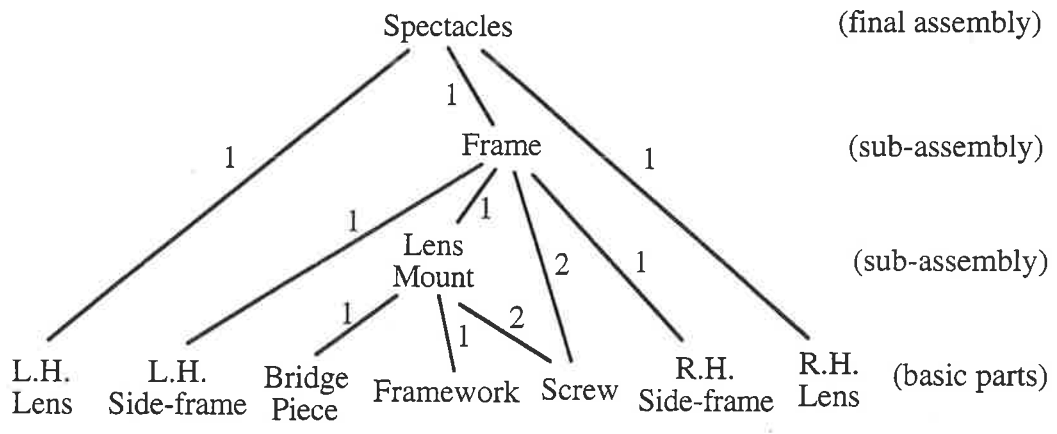

FIGURE 5.6.1: STRUCTURE OF A PAIR OF SPECTACLES

A typical Bill of Materials Problem is to compute the number of basic parts needed to construct a given number of final assemblies, e.g., given the need for 10 pairs of spectacles, to deduce that 40 screws are required, and so on. This can be done by a downward pass over the graph. Other problems, such as finding the total cost of materials used, require an upward pass. Still others require a depth first traversal, and so on.

The Chart of Accounts Problem is somewhat simpler. A Profit and Loss Statement or similar accounting document has major headings such as 'Revenue' and 'Expenditure', which may be subdivided into categories such as 'Capital' and 'Recurrent', which are in turn subdivided, and so on, down to basic accounts. The structure is an ordered rooted tree, rather than a general acyclic graph. Given the balances for the basic accounts, the totals for the major headings may be computed by an upward pass from the leaves to the root. (A complication in practice is that the same set of basic accounts may be formed into different trees for different accounting purposes.)

How do such structures affect system design? Consider the parts requirements problem just outlined, where, given a required number of final assemblies, it is desired to find the required numbers of each sub-assembly and basic part. Since a requirement for a given sub-assembly generates new requirements for lower level sub-assemblies, a simple-minded specification of the problem gives an SDG based on Figure 5.6.1. Since the graph is acyclic, it could become the basis of an implementation. The problem with any graph based on Figure 5.6.1 is that it is not compatible with a basic assumption of the Canonical Decomposition Method, that a system is a *fixed* network of component processes. Any event that changed the parts structure would require a corresponding *dynamic* change to the system network. This is outside the scope of the thesis.

```
use Schedule;
type part is private;
package Bill_of_Materials is
   procedure Requirements (Required : array (part) of natural);
end Bill_of_Materials;
package body Bill_of_Materials is
   Uses : array (part, part) of natural := (others => (others => 0));
   procedure Requirements (Required : array (part) of natural) is
      Inherited : array (part) of natural := (others => 0);
      Generated : array (part) of natural := (others => 0);
      Done : boolean := false;
   begin
      all p in part loop
         Inherited(p) := Required(p);
      end loop;
      while not Done loop
         Done := true;
         all major in part loop
            if Inherited(major) > 0 then
               Schedule.Requirement(major, Inherited(major));
               all minor in part loop
                  if Uses(major, minor) > 0 then
                     Generated(minor) :=
                        Generated(minor)+Uses(major, minor)*Inherited(major);
                     Done := false;
                  end if;
               end loop;
            end if;
         end loop;
         all p in part loop
            Inherited(p) := Generated(p);
            Generated(p) := 0;
         end loop;
      end loop;
   end Requirements;
end Bill_of_Materials;
```

EXAMPLE 5.6.1: CALCULATING REQUIREMENTS FOR PARTS

To make progress, it is first necessary to specify the problem in iterative form, as in Example 5.6.1. An outer **while** loop propagates requirements down one level at each iteration. The number of times part 'major' incorporates part 'minor' is given by 'Uses(major, minor)', which may be thought of as a matrix. 'Inherited' contains the requirements from the previous level, and 'Generated' accumulates the requirements for the next level. The inherited requirements at each level are sent to the external 'Schedule' package for whatever purpose it needs them. The number of iterations of its **while** loop is unpredictable, being determined by the maximum path length in the parts structure. Likewise, the system topology is determined by the number of levels in the parts structure and cannot be described by a fixed process network. However, it is possible to design a fixed network of processes to implement each iteration of the loop body. Figure 5.6.2 shows the SDG for the first part of the loop body. The final values of 'Generated' for one iteration become the initial values of 'Inherited' for the next iteration.

Independence

The SDG shows that the values of 'Inherited' may be inspected independently, by part. Each part that has inherited a non-zero requirement may generate a thread for each part it uses, so the inspection of each element of 'Uses' may proceed independently too. Finally, the accesses to 'Generated' may also be independent. However, full independence is not possible unless all three processes are separated. Combining the 'Inherited' and 'Generated' processes to complete the loop would destroy the independence because there is a data flow from each part to the components from which it is assembled. This is implied by the marked edge in Figure 5.6.2, which indicates that the index of 'Generated' and the first index of 'Uses' differ, and the potential many-one agreement is absent. (Considering 'Uses' as a matrix, transposing it would remove this conflict, but would create a new conflict between 'Uses' and 'Inherited'.)

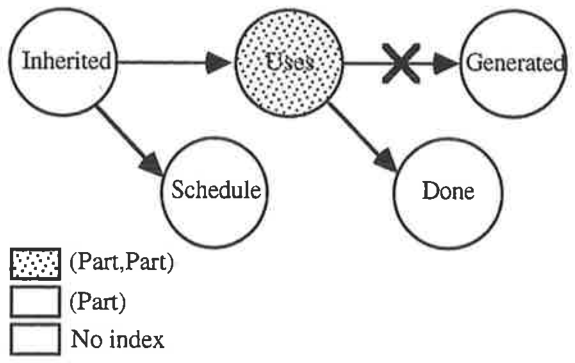

FIGURE 5.6.2: ONE ITERATION OF THE 'REQUIREMENTS' PROCEDURE

A variation of the procedure of Example 5.6.1 is possible when a parts graph has a fixed number of levels. Each part can be assigned a level number given by the length of the longest path connecting it to a leaf. This allows the **while** loop in Example 5.6.1 parts to be unwound into a fixed number of copies, and the parts data to be partitioned by level. The resulting SDG then becomes essentially a number of copies of Figure 5.6.2 laid end to end, starting with final assemblies and finishing with basic parts. Given this fixed partitioning, the processing of the lower level partitions may lag behind that of the higher levels by successively increasing amounts. It becomes possible to mix batches of 'Requirements' events with other kinds of event and still preserve real time equivalence. A drawback of using fixed levels is that the resulting process network cannot handle an event that increases the level number of a part. Its record would have to be moved against the data flow.

In summary, when the dependences between elements form an acyclic graph, it is likely that the system specification will include events that access the structure recursively or iteratively. By considering one iteration of such an event in isolation, the methods outlined for simpler kinds of event may be applied to yield a process graph that allows independent access. However, since this requires the data to be partitioned into levels, it requires a non-trivial change to the specification, which must be assumed to be the responsibility of the specifier. Consequently, this topic will not be discussed further, and acyclic dependences between elements of an array will be treated in the same way as cyclic ones.

6. Requirements Analysis

The emphasis of this chapter is the interaction between requirements analysis and finding an efficient system implementation. We have established that every specification has a unique CPG, which offers a restricted range of possible implementations, none of which may be particularly efficient. However, it is sometimes possible to change the specification of a problem so that a different range of solutions results. Surprisingly, this may be possible without changing the external behaviour of the system. Even when the behaviour of the system must be changed to find an efficient implementation, the new behaviour may be equally acceptable to the user. In practice, such negotiation about specifications often takes place between a systems analyst and a client. The aim of this chapter is to show why such negotiations are useful.

6.1 An Order-Processing System

A business sells products to customers. An 'Order' is a request from a customer to buy a certain quantity of a product. The business will send the full quantity if it can, otherwise it will send what it has (the 'Stock'). When there is a shortage, it is first-come, first-served: the available stock is sent to the first customer to order it. Each customer owes a 'Balance', which must be increased by the value of the items actually delivered, rather than the value ordered. There is no limit to how much a customer may owe. Orders made by non-existent customers or for non-existent products must be ignored.

Is there an efficient solution to this problem involving only sorting and sequential access, avoiding random access?

The first step is to formalise the requirement as a specification, as in Example 6.1.1. The specification uses two external packages: 'Invoice' prints a properly formatted invoice on the basis of the parameters it is given, and 'Error' displays error diagnostics. In practice, 'Invoice' will need to access attributes such as customer addresses and product descriptions, and so on, but these attributes are omitted from the example for the sake of simplicity. The specification introduces two variables not explicitly mentioned in the informal specification: 'Authorised', which indicates whether a given customer identifier is that of a valid account, and 'Offered', which indicates whether a given product code refers to an actual product that is offered for sale. These 'existence variables' would probably be implicit in a database, being true for those primary keys with a corresponding row, and false for those without.

The second step towards a solution is to draw the SDG of Figure 6.1.1. White vertices represent the attributes of the customer table, and grey vertices represent those of the product table. Black vertices represent inputs and outputs. However, for simplicity, local variables are not shown.

```
with Error, Invoice;
generic
   type customer is private;
   type product is private;
package Order_Processing is
   subtype money is integer;
   procedure Order (Who : customer; What : product; Qty_Ordered : positive);
   -- other event specifications
package body Order_Processing is
   Authorised : array (customer) of boolean := (others => false);
   Balance : array (customer) of money := (others => 0);
   Offered : array (product) of boolean := (others => false);
   Price : array (product) of money := (others => 0);
   Stock : array (product) of natural := (others => 0);
   procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
      Qty_Delivered : natural := 0;
      Value_Ordered : money := 0;
   begin
      if not Offered (What) then
         Error.Product (What);
      elsif not Authorised (Who) then
         Error.Customer (Who);
      else
         Value_Ordered := Price (What) * Qty_Ordered;
         Qty_Delivered := min (Stock (What), Qty_Ordered);
         Invoice.Deliver (Who, What, Qty_Delivered);
         Stock (What) := Stock (What) – Qty_Delivered;
         Balance (Who) := Balance (Who) + Qty_Delivered * Price (What);
      end if;
   end Order;
   -- other event procedures
end Order_Processing;
```

EXAMPLE 6.1.1: SPECIFYING AN 'ORDER'

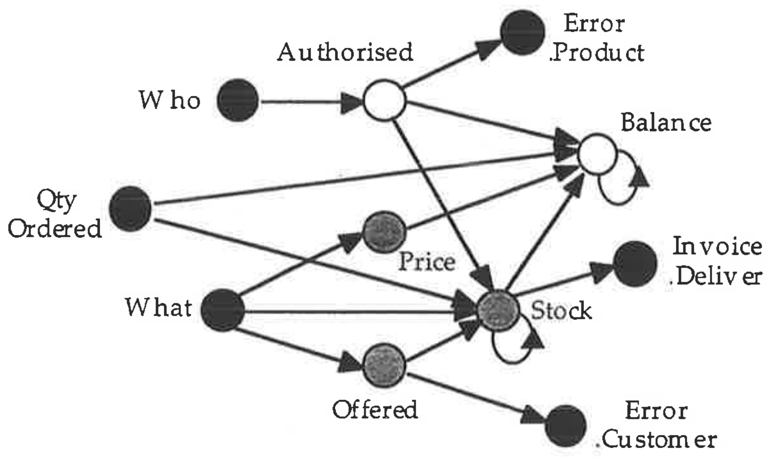

FIGURE 6.1.1: THE SDG FOR AN ORDER

The edges of the graph are drawn as follows:

- There is an edge from 'Who' to 'Authorised' because 'Who' is used to select the customer row whose existence is tested.
- There is an edge from 'What' to 'Offered' because 'What' is used to select the product row whose existence is tested.
- There is an edge from 'Qty Ordered' to 'Stock' because it is typically the amount by which the stock changes.
- There is an edge from 'Qty Ordered' to 'Balance' because it affects the amount by which the customer's balance changes.
- There is an edge from 'Price' to 'Balance', because the change in the customer's balance depends on the price of the product.
- There is an edge from 'Stock' to 'Balance', because the quantity actually sold to the customer depends on whether there is sufficient stock.
- There are edges from 'Stock' and 'Balance' to themselves, because their values after the order event depend on their own values before.
- There are edges from 'Authorised' and 'Offered' to both 'Balance' and 'Stock' because their values are changed only if the customer and product rows exist.
- There is an edge from 'Stock' to 'Invoice.Deliver' because it affects 'Qty Delivered', which is one of its parameters.
- There are edges from 'Authorised' to 'Error.Product' and from 'Offered' to 'Error.Customer', because an error should be reported if either row is missing.

Several edges have been omitted from Figure 6.1.1 because they do not affect its transitive closure; for example, no edge was drawn from 'Offered' to 'Balance', because there is already a path via 'Stock'. Figure 6.1.1 is a transitive reduction of the full graph. There are no conflicts other than those implied by the colours of the vertices.

Input and output vertices carry very little design information. Essentially, all inputs must be sources, and all outputs must be sinks. However, there is some useful information present, for example, none of the outputs depends on 'Balance'. This means that it might be possible to update 'Balance' less frequently than the outputs are produced. For example, invoices might be produced daily, but balances might only need to be updated once per month. Omitting the input and output vertices, and the loops on 'Balance' and 'Stock', leads to the much simpler SDG of Figure 6.1.2.

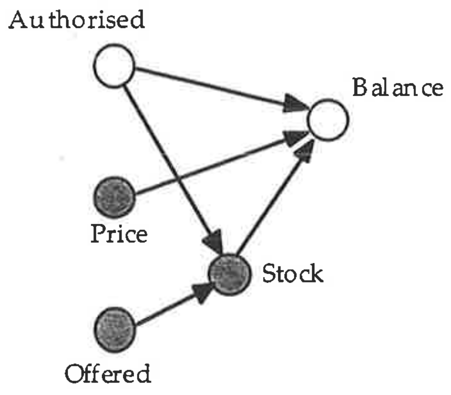

FIGURE 6.1.2: THE SIMPLIFIED SDG

Since Figure 6.1.2 is acyclic, it may also be interpreted as the CPG. Each vertex is a separate trivial strong component, and represents a minimal process. Vertices in Figure 6.1.2 therefore map directly to processes that access the database attributes named by their labels. The edges of Figure 6.1.2 map to the queues that carry data between them.

The next problem is to find the best way to cluster the minimal processes into composite processes, which become the components of the finished design. In this example, the design domain is assumed to be a batch information system that uses sequential access.

6.2 Process Composition

Combining two processes avoids the cost of transferring data between them, but it may increase the cost of accessing the database. If two attributes with the same index are accessed or updated by the same composite process, they can be present in the same row of a table, which can therefore be accessed once instead of twice. But if they have different indices, they cannot be present in the same row, and independent access would become impossible. Attributes with different domains certainly have different indices. It is therefore assumed that two steps should be combined if they access the same index, but should be kept separate if their keys are different. In Figure 6.1.2, it is only useful to combine vertices with the same colour.

Figure 6.2.1 shows how the attributes of Figure 6.1.2 can best be grouped. Each ellipse contains vertices that have the same indices, which allows sequential access.

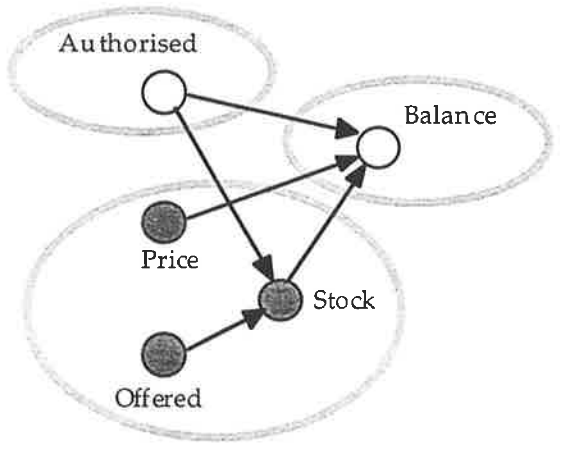

FIGURE 6.2.1: THE BEST GROUPING OF ATTRIBUTES

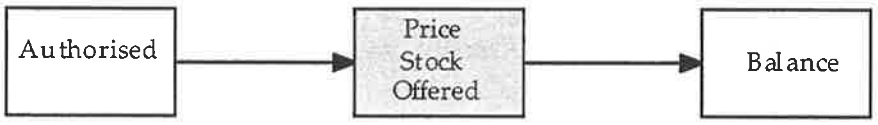

FIGURE 6.2.2: THE COMPOSITE PROCESS GRAPH

Figure 6.2.2 shows the resulting process graph. The steps implement the following functions:

Step 1: The orders are sorted and matched sequentially against the customer table. Any orders for which the customer row is missing are in error.

Step 2: The orders are sorted and matched sequentially against the product table. If there is no corresponding row in the table, the order is in error. Otherwise, the quantity delivered is the lesser of the quantity ordered and the stock, and the stock is decreased accordingly. The price and quantity delivered are transmitted to the third step.

Step 3: The orders are sorted into customer sequence again, and matched against the customer table. The customer balances are incremented by the product of price and quantity delivered.

It is sensible to ask if Step 1 and Step 3 could be combined, as this would enable the two accesses of the customer table to be reduced to one. But this is impossible. Step 1 must precede Step 2; it would be a mistake to decrement the stock in response to an order for an unauthorised customer. Likewise, Step 2 must precede Step 3, as the change in a customer's balance depends on the stock of the product. Combining Steps 1 and 3 would result in a composite process that would have to simultaneously precede and follow Step 2, which is clearly impossible. It would create a cyclic pipe-line between processes, contrary to the Data Flow Theorem.

Provided they are only accessed by the processes to which they are assigned, it does not matter whether or not attributes with a common key belong to the same table. In this example, 'Authorised' and 'Balance' could be held in two separate tables, or in a single table. The issue is a minor space-time trade-off. Keeping the attributes in separate tables means that the 'customer' keys must be stored twice. Placing them in the same table means that, although the keys are stored only once, 'Balance' has to be retrieved uselessly in the first process, and 'Authorised' has to be retrieved uselessly in the third process.

6.3 Adding Other Kinds of Event

```
with Error, Invoice;
generic
   type customer is private;
   type product is private;
package Order_Processing is
   subtype money is integer;
   procedure Open (Who : customer);
   procedure Close (Who : customer);
   procedure Pay (Who : customer; Amount : money);
   -- other event specifications
package body Order_Processing is
   Authorised : array (customer) of boolean := (others => false);
   Balance : array (customer) of money := (others => 0);
   Offered : array (product) of boolean := (others => false);
   Price : array (product) of money := (others => 0);
   Stock : array (product) of natural := (others => 0);
   procedure Open (Who : customer) is
   begin
      if not Authorised (Who) then
         Authorised (Who) := true;
         Balance (Who) := 0;
      end if;
   end Open;
   procedure Close (Who : customer) is
   begin
      if not Balance (Who) /= 0 then
         Authorised (Who) := false;
      end if;
   end Close;
   procedure Pay (Who : customer; Amount : money) is
   begin
      if Authorised (Who) then
         Balance (Who) := Balance (Who) – Amount;
      end if;
   end Close;
   -- other event procedures
end Order_Processing;
```

EXAMPLE 6.3.1: SPECIFYING 'OPEN' 'CLOSE' AND 'PAY' EVENTS.

A real-life order processing system should provide for many kinds of event. It should be able to open new customer accounts, close them, change names and addresses, accept payments, add or remove products, adjust their prices, and record their deliveries. These events may be analysed in the same way as the order event, and their new dependences must be added to the existing SDG; a system design must simultaneously provide data flows for all the kinds of event. As examples, Example 6.3.1 shows the specifications of the 'Open', 'Close' and 'Pay' events.

The 'Open' and 'Pay' events (and some other possible events, not specified in Example 6.3.1) involve dependences (e.g., to 'Balance' from 'Authorised') that are already present in Figure 6.1.2, so the design suggested by Figure 6.2.2 can handle all of them without modification. It turns out that only one kind of event causes a new constraint, and that is the 'Close' event. Because of the common-sense rule that the account should be left open if the customer's balance is non-zero, the new value of 'Authorised' depends on the existing value of 'Balance', as shown in Figure 6.3.1. Adding this dependence to the graph of Figure 6.1.2 results in the graph of Figure 6.3.2, which has a strong component comprising 'Authorised', 'Balance' and 'Stock', i.e., there is a directed path between any pair of them.

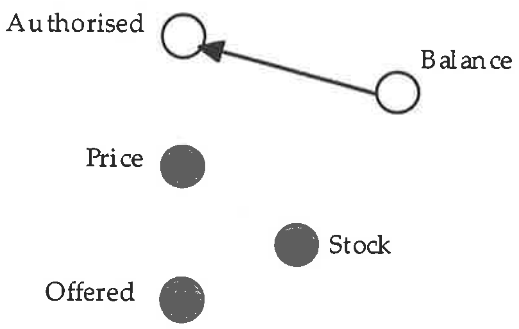

FIGURE 6.3.1: A DEPENDENCE DUE TO 'CLOSE' EVENTS

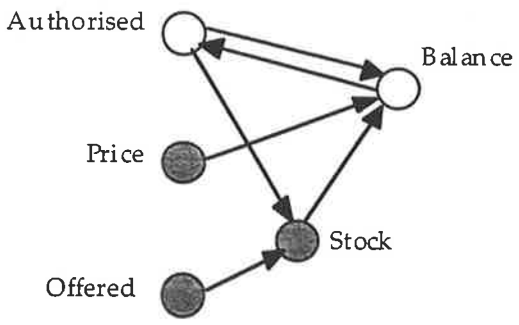

FIGURE 6.3.2: A CYCLE DUE TO 'CLOSE' EVENTS

Since the all the vertices of a strongly-connected component form one separable process, the CPG must group the vertices as shown in Figure 6.3.3.

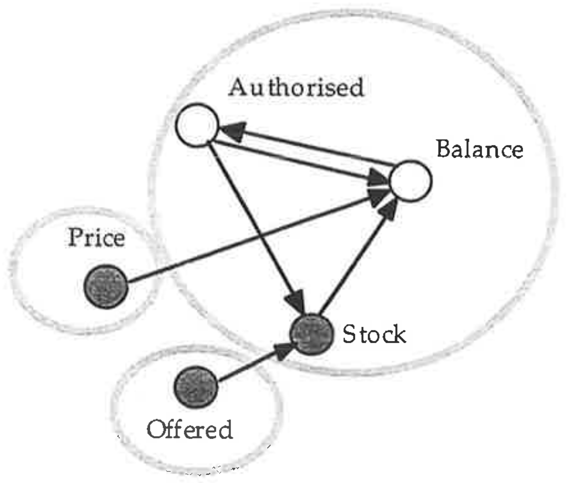

FIGURE 6.3.3: SEPARABLE PROCESSES WITH 'CLOSE' EVENTS

There are now only three separable processes; 'Authorised', 'Balance' and 'Stock' must be accessed in the same one. This is easy to understand. If they had been accessed in separate steps, no one step could precede either of the others in the data flow. So all three attributes must be accessed at the same time. Because this composite process involves attributes with different keys, it cannot use sequential access, and must use less efficient random access. (If the events were sorted into the order of the rows of one table to avoid one random access, this would scramble the sequence of events affecting a given row of the other table.) The presence of a cycle in the SDG is harmless in itself; it is the fact that the cycle involves attributes with different domains that causes the problem.

'Offered' and 'Price' could be accessed sequentially, in a step preceding the random access of the other three attributes. But this would actually be an inferior design. Since the random-access process must access the same rows of the product table to update 'Stock', it can access 'Offered' and 'Price' at the same time at no additional cost. So the best design is a single random-access step. The cycle introduced by the 'Close' event completely invalidates the efficient sequential access design of Figure 6.2.2.

(This analysis does not exclude the possibility that 'Authorised' and 'Balance' are stored remotely from 'Stock'. It means that the processes that access them have to be closely coupled. Neither does it exclude parallelism, provided that a suitable locking protocol is enforced.)

6.4 Avoiding Data-Dependence Cycles

Given the order of magnitude difference in efficiency that usually exists between sequential access and random access, a clever systems analyst would certainly attempt to remove the cycle from the graph of Figure 6.3.3.

A common strategy used in information system design is to partition events into two or more 'modes'. Each mode is then implemented using a different design. We have seen that the graph of Figure 6.1.2 can be implemented efficiently, and there is certainly no difficulty in implementing Figure 6.3.1, which simply involves an update of the 'customer' attributes. To be useful, the partitioning must assign at least one of the edges in the cycle of Figure 6.3.3 to a different mode from the others. Here, this can only be done by separating the 'Close' and 'Order' events.

But this solution has drawbacks. If there are some 'Order' events followed in time by a 'Close' event, followed by further 'Order' events, correct results can only be guaranteed by processing the first set of orders, then processing the 'Close' event, then the remaining orders. This limits the size of batches, and reduces the efficiency of the system. If the batches are to be economically large, 'Close' events would either need to be rare, or capable of being deferred until a large enough batch of orders had been accumulated. If the intention of 'Close' events is to prevent delinquent customers making further orders, then delay is clearly inadvisable — although presumably 'Close' events should not occur very often.

In practice, many systems are decomposed into modes in a rather arbitrary way, separating 'processing' from 'file maintenance'. Thus all events that affect only 'customer' information might be grouped together, e.g., opening new customer accounts, closing accounts, changing names and addresses, and so on. Similarly, another file maintenance mode might update the 'product' information. Only events that involve both files, such as 'Order' events, would be handled by the 'processing' mode. This conventional arrangement is clearly inferior, as it reduces the sizes of batches; the batch of 'Order' events has to be processed before every file maintenance operation, and conversely, the batches of file maintenance events have to be processed before every batch of orders. Sometimes a virtue can be made out of necessity, for example, by declaring that file maintenance operations have higher priority than other processing, so that file maintenance should logically be done before processing every batch of orders, but this does not really preserve real-time equivalence.

An alternative is for the systems analyst to negotiate a harmless change to the specification to remove the cycle from the SDG. One option here is to introduce a new boolean attribute, 'Closed', that is changed without reference to 'Balance', as in Example 6.4.1. Once a customer account is closed, no further orders will be accepted, but payments made by the customer will still be allowed. The modified event specifications are shown in Example 6.4.1, and the resulting SDG is shown in Figure 6.4.1.

The revised specification is probably closer to the client's intention of what a 'Close' event should do anyway. Since the SDG of Figure 6.4.1 is acyclic, it also leads to an efficient design. After an account has been closed, and payments reduce its balance to zero, its row can be removed from the table — it becomes a garbage collection problem.

```
with Error, Invoice;
generic
   type customer is private;
   type product is private;
package Order_Processing is
   subtype money is integer;
   procedure Open (Who : customer);
   procedure Close (Who : customer);
   procedure Pay (Who : customer; Amount : money);
   -- other event specifications
package body Order_Processing is
   Authorised, Closed : array (customer) of boolean := (others => false);
   Balance : array (customer) of money := (others => 0);
   Offered : array (product) of boolean := (others => false);
   Price : array (product) of money := (others => 0);
   Stock : array (product) of natural := (others => 0);
   procedure Open (Who : customer) is
   begin
      if not Authorised (Who) then
         Authorised (Who) := true;
         Balance (Who) := 0;
      end if;
   end Open;
   procedure Close (Who : customer) is
   begin
      if not Authorised (Who) then Error.Customer (Who);
      else Closed (Who) := true;
      end if;
   end Close;
   procedure Pay (Who : customer; Amount : money) is
   begin
      if not Authorised (Who) then Error.Customer (Who);
      else Balance (Who) := Balance (Who) – Amount;
      end if;
   end Close;
   procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
      Qty_Delivered : natural := 0;
      Value_Ordered : money := 0;
   begin
      if not Offered (What) then Error.Product (What);
      elsif not Authorised (Who) then Error.Customer (Who);
      elsif Closed (Who) then Error.Closed (Who);
      else
         Value_Ordered := Price (What) * Qty_Ordered;
         Qty_Delivered := min (Stock (What), Qty_Ordered);
         Invoice.Deliver (Who, What, Qty_Delivered);
         Stock (What) := Stock (What) – Qty_Delivered;
         Balance (Who) := Balance (Who) + Qty_Delivered * Price (What);
      end if;
   end Order;
-- other event procedures
end Order_Processing;
```

EXAMPLE 6.4.1: SPECIFYING ‘OPEN’ ‘CLOSE’ AND ‘PAY’ EVENTS.

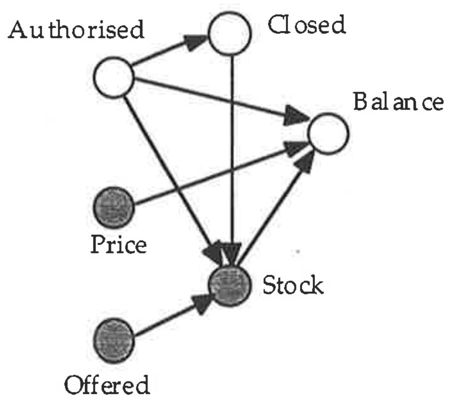

FIGURE 6.4.1: SDG WITH A 'CLOSED' ATTRIBUTE

However, the original problem hasn't really disappeared; checking the conditions for garbage collection would have to be done by a separate mode. However, since the only function of this mode is to reclaim storage space, it could have very low priority. Such modes are common in practice, and are called 'file weeding'.

6.5 Sensitivity of Design to Specification

The preceding section demonstrated that adding requirements for additional events could cause a dependence cycle, but even a single event can cause one.

Consider an extension of the original order processing problem to take account of a customer's credit-worthiness. Each customer account has a 'Credit Limit' that its balance may never exceed. For an order to be accepted, the value of the order should not exceed the customer's 'Available Credit', defined as the difference between the customer's 'Credit Limit' and existing 'Balance'. The modified event is specified in Example 6.5.1, and the resulting SDG is shown in Figure 6.5.1.

There is an edge in Figure 6.5.1 from 'Stock' to 'Balance' because the stock determines the value that can be delivered to the customer, and therefore the amount by which 'Balance; must be increased. There is an edge from 'Balance' to 'Stock' because the existing balance determines whether an order is accepted. There is a cycle between 'Stock' and 'Balance' that even the most ingenious designer cannot remove. It is impossible to place its edges in different modes because they arise from the same event. There is no way to avoid random access.

This example illustrates a common experience of anyone who has had to maintain batch information systems. Changes to specifications are constantly being made, and many of them are easily incorporated into the existing system. But occasionally, a 'simple' change is requested that makes it necessary to redesign the system from scratch. Such a change is one that creates a new cycle in the SDG.

```
with Error, Invoice;
generic
   type customer is private;
   type product is private;
package Order_Processing is
   subtype money is integer;
   procedure Order (Who : customer; What : product; Qty_Ordered : positive);
   -- other event specifications
package body Order_Processing is
   Authorised : array (customer) of boolean := (others => false);
   Balance, Credit_Limit : array (customer) of money := (others => 0);
   Offered : array (product) of boolean := (others => false);
   Price : array (product) of money := (others => 0);
   Stock : array (product) of natural := (others => 0);
   procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
      Qty_Delivered : natural := 0;
      Value_Ordered, Available_Credit : money := 0;
   begin
      if not Offered (What) then
         Error.Product (What);
      elsif not Authorised (Who) then
         Error.Customer (Who);
      else
         Value_Ordered := Price (What) * Qty_Ordered;
         Available_Credit := Credit_Limit (Who) – Balance (Who);
         if Value_Ordered > Available_Credit then
            Error.Credit (Who, What, Value_Ordered, Available_Credit);
         else
            Qty_Delivered := min (Stock (What), Qty_Ordered);
            Invoice.Deliver (Who, What, Qty_Delivered);
            Stock (What) := Stock (What) – Qty_Delivered;
            Balance (Who) := Balance (Who) + Qty_Delivered * Price (What);
         end if;
      end if;
   end Order;
   -- other event procedures
end Order_Processing;
```

EXAMPLE 6.5.1: SPECIFYING AN 'ORDER' WITH A CREDIT LIMIT

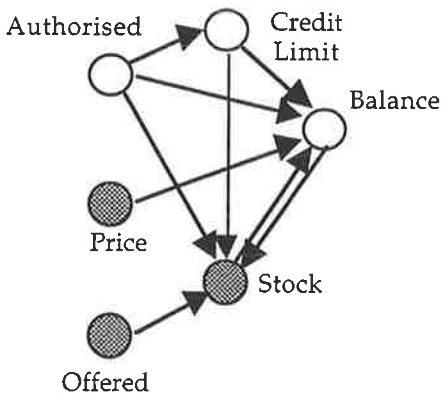

FIGURE 6.5.1: ORDERS WITH A CREDIT LIMIT

```
with Error, Invoice;
generic
   type customer is private;
   type product is private;
package Order_Processing is
   subtype money is integer;
   procedure Order (Who : customer; What : product; Qty_Ordered : positive);
   -- other event specifications
package body Order_Processing is
   Authorised : array (customer) of boolean := (others => false);
   Balance, Credit_Limit, Commitment : array (customer) of money := (others=> 0);
   Offered : array (product) of boolean := (others => false);
   Price : array (product) of money := (others => 0);
   Stock : array (product) of natural := (others => 0);
   Back_Order : array (customer, product) of natural := (others => (others => 0));
   procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
      Qty_Delivered, Shortage : natural := 0;
      Value_Ordered, Available_Credit : money := 0;
   begin
      if not Offered (What) then
         Error.Product (What);
      elsif not Authorised (Who) then
         Error.Customer (Who);
      else
         Value_Ordered := Price (What) * Qty_Ordered;
         Available_Credit := Credit_Limit (Who)
                                             – (Balance (Who) + Commitment (Who));
         if Value_Ordered > Available_Credit then
            Error.Credit (Who, What, Value_Ordered, Available_Credit);
         else
            Qty_Delivered := min (Stock (What), Qty_Ordered);
            Shortage := Qty_Ordered – Qty_Delivered;
            Invoice.Deliver (Who, What, Qty_Delivered);
            Stock (What) := Stock (What) – Qty_Delivered;
            Balance (Who) := Balance (Who) + Qty_Delivered * Price (What);
            Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
            Commitment (Who) := Commitment (Who) + Shortage * Price (What);
         end if;
      end if;
   end Order;
   -- other event procedures
end Order_Processing;
```

EXAMPLE 6.6.1: SPECIFYING AN 'ORDER' WITH A COMMITMENT

A further example shows that a design can be affected even by how facts are represented in the database. Consider the 'Order' event yet again, but now assume that when the stock is too low, unfulfilled orders are automatically placed on back order, i.e., a promise is made to fill the shortage as soon as new supplies arrive. The customer is therefore committed to pay for these items in the future, which should be taken into account in assessing credit-worthiness. The total amount of future commitment is represented by the attribute 'Commitment'. The modified event is specified in Example 6.6.1, and the resulting SDG is shown in Figure 6.6.1. 'Back

Order', which is number of items put on back order, has a composite key comprising the customer and the product.

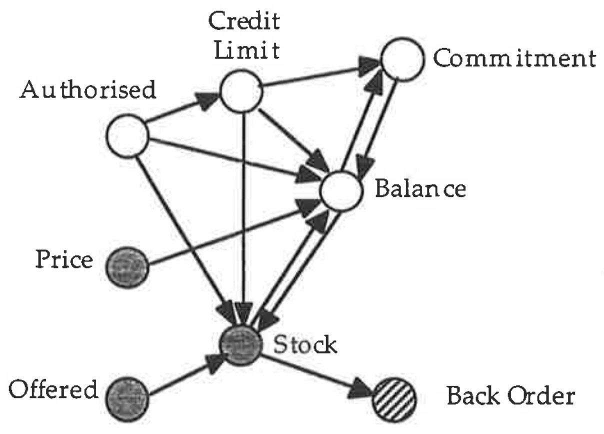

FIGURE 6.6.1: ORDERS WITH COMMITMENTS

The SDG now has a strong component containing 'Stock', 'Balance' and 'Commitment'. 'Stock' affects how the value ordered should be added to 'Balance' and 'Commitment', and 'Balance' and 'Commitment both affect whether an order is accepted, and therefore determine whether 'Stock' should be updated. There appears to be little prospect of finding a solution that permits sequential access to be used.

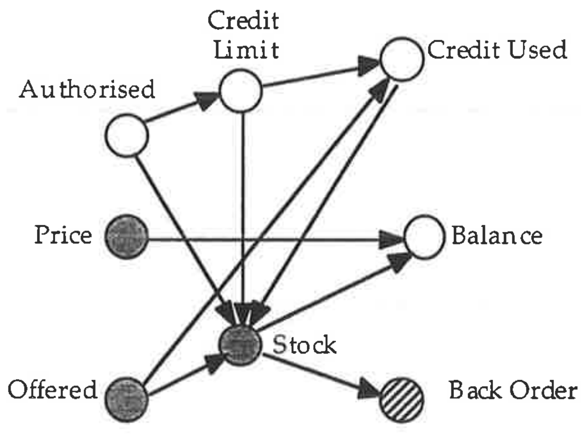

FIGURE 6.6.2: ORDERS WITH CREDIT USED

But here's the trick! Suppose that the data representation is changed. It is the *sum* of 'Balance' and 'Commitment' that determines whether an order is acceptable. Instead of 'Commitment', why not store the sum itself, calling it 'Credit_Used'? If needed, it would still be possible to determine 'Commitment' by subtracting 'Balance' from 'Credit Used'. The revised specification is shown if Example 6.6.2. 'Credit Used' is the amount that the customer must pay either now or in the future; it depends on the value ordered but is independent of 'Stock'. The resulting graph, shown in Figure 6.6.2, has no cycles, it may be optimised by clustering its separable processes as in Figure 6.6.3, and may be implemented by a sequence of five sequential steps, as suggested in Figure 6.6.4. (Step 4 and Step 5 could be processed in either sequence.)

```
with Error, Invoice;
generic
   type customer is private;
   type product is private;
package Order_Processing is
   subtype money is integer;
   procedure Order (Who : customer; What : product; Qty_Ordered : positive);
end Order_Processing;
package body Order_Processing is
   Authorised : array (customer) of boolean := (others => false);
   Balance, Credit_Used, Credit_Limit : array (customer) of money := (others => 0);
   Offered : array (product) of boolean := (others => false);
   Price : array (product) of money := (others => 0);
   Stock : array (product) of natural := (others => 0);
   Back_Order : array (customer, product) of natural := (others => (others => 0));
   procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
      Qty_Delivered, Shortage : natural := 0;
      Value_Ordered, Available_Credit : money := 0;
   begin
      if not Offered (What) then Error.Product (What);
      elsif not Authorised (Who) then Error.Customer (Who);
      else Value_Ordered := Price (What) * Qty_Ordered;
         Available_Credit := Credit_Limit (Who) – Credit_Used (Who);
         if Value_Ordered > Available_Credit then
            Error.Credit (Who, What, Value_Ordered, Available_Credit);
         else
            Qty_Delivered := min (Stock (What), Qty_Ordered);
            Shortage := Qty_Ordered – Qty_Delivered;
            Invoice.Deliver (Who, What, Qty_Delivered);
            Stock (What) := Stock (What) – Qty_Delivered;
            Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
            Balance (Who) := Balance (Who) + Qty_Delivered * Price (What);
         end if;
         Credit_Used (Who) := Credit_Used (Who) + Value_Ordered;
      end if;
   end Order;
end Order_Processing;
```

EXAMPLE 6.6.2: SPECIFYING AN 'ORDER' WITH CREDIT USED

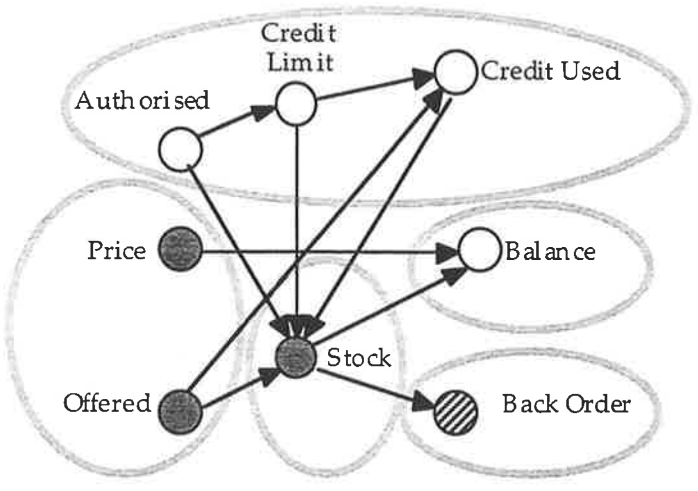

FIGURE 6.6.3: COMPOSITE PROCESSES FOR ORDERS WITH CREDIT USED

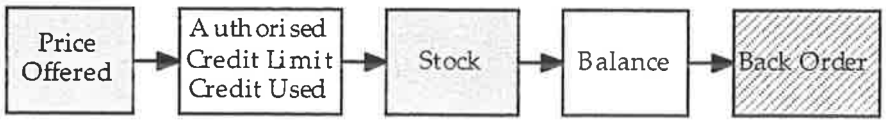

FIGURE 6.6.4: PROCESS GRAPH FOR ORDERS WITH CREDIT USED

Figure 6.6.4 represents the following design solution:

Step 1: Sort the orders into product sequence and match them against the product table. Check that the product row exists, and find its price.

Step 2: Sort the orders into customer sequence and match them against the customer table. Check that the customer row exists, decide if the credit limit will be exceeded, and update the customer's credit used according to the price and quantity ordered.

Step 3: Sort the orders into product sequence and match them against the product table. Decide what quantity can be delivered, and update the stock.

Step 4: Sort the orders into customer sequence and match them against the customer table. Update the customer's balance according to the price and quantity delivered.

Step 5: Sort the orders into customer, product sequence and match them against the back order table. Update the back order quantity according to the difference between the quantity ordered and the quantity delivered.

Although five sequential access processes are needed, it is likely that this design will still prove much more efficient than one that uses random access.

This example illustrates a surprising fact: even if the data representation is a correctly normalised schema, the correct choice of representation can make the system more efficient. In this example, replacing 'Commitment' by 'Credit Used' moved an assignment outside an **if** statement and therefore eliminated some dependences.

It is not easy to generalise this result. Choosing the best set of attributes seems to call for genuine creativity. On the other hand, it is easy to see how to make a data representation worse. It is merely necessary to find two attributes that are accessed in different processes and pack their contents into one variable. For example, two boolean variables might be combined into one 4-valued variable. The effect on the SDG would be to merge two vertices into one, perhaps creating a cycle where none existed before. Exactly this effect could be achieved in Figure 6.4.1 by packing together 'Authorised' and 'Closed'. Indeed, there is a temptation to do so; they could be replaced by a single variable with three states: 'Unused', 'Authorised' and 'Closed'. Presumably, successful design must consist of factorising variables into independent parts — although this does not really seem to apply to the 'Order' event example, where 'Credit Used' is just a linear combination of 'Balance' and 'Commitment'.

6.7 Optimistic Fail-Safe Systems

Example 6.5.1 specified a form of the order processing problem that had no efficient real-time equivalent implementation. This sad fact resulted from a cycle between 'Stock' and 'Balance' in the SDG of an 'Order' event. It is necessary to check 'Balance' before adjusting 'Stock' to ensure that goods are not issued to customers who cannot afford them. It is necessary to check 'Stock' before adjusting 'Balance' to ensure that customers are not billed for goods that cannot be supplied. The only way this can be done in a real-time equivalent system is to access both attributes in the same process. Since they have different indices, this destroys the opportunity to use independent access.

But what if the design is not real-time equivalent? Is it possible to negotiate with the client a specification that has an efficient implementation? To be efficient, the design would have break the cycle between 'Stock' and 'Balance'. One way to do this would be to separate the inspection of 'Balance' from the updating of 'Balance', or separate the inspection of stock from the updating of stock, or separate both.

What happens if the inspection and updating of 'Balance' are separated, as in Figure 6.7.1? The processing of an order would then consist of checking that the customer could afford the order, finding the amount that could be delivered, then updating the customer's balance owing. This would be faulty because it is always the initial value of 'Balance' that is inspected. A customer might make a series of small orders, none of which individually would exceed the customer's credit limit. However, they might do so cumulatively. Allowing a customer to exceed their credit limit is a potentially dangerous situation. The client would almost certainly reject such a solution.

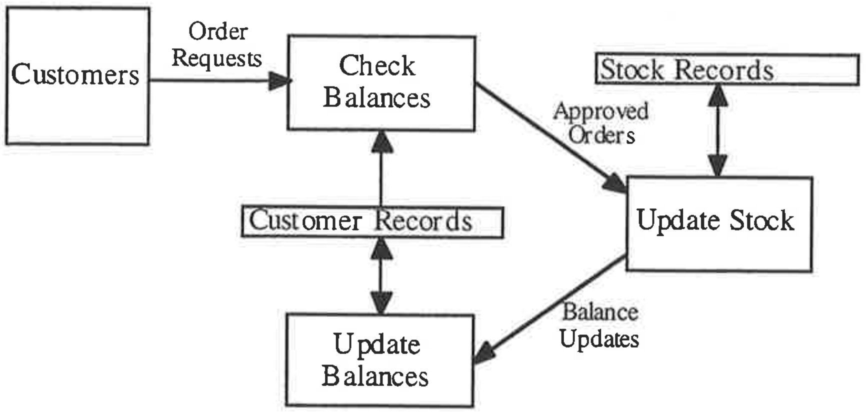

FIGURE 6.7.1: SEPARATING INSPECTION AND UPDATING OF 'BALANCE'

The fault in Figure 6.7.1 arises because 'Balance' is not updated soon enough. Figure 6.7.2 attempts to redress this by updating 'Balance' before updating 'Stock'. Since it is impossible to know whether the full amount of an order can be delivered until 'Stock' is inspected, 'Balance' can only be updated under the assumption that the whole order will be satisfied, i.e., the customer balance is debited the maximum possible amount. There is no longer any danger of customers exceeding their credit limits. On the other hand, if updating 'Stock' reveals a shortage, the value assigned to 'Balance' will be incorrect, the customer having been billed for goods that can't be supplied. Therefore, the 'Adjust Balances' process is used to credit customers for any goods that were ordered but not supplied.

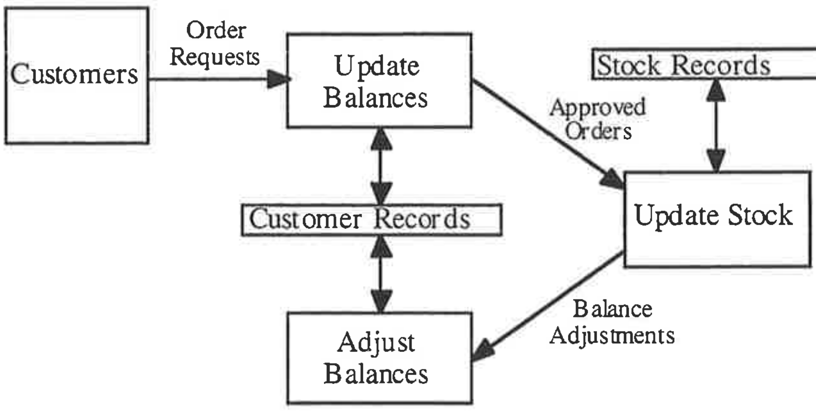

FIGURE 6.7.2: OPTIMISTIC UPDATING OF 'BALANCE'

The resulting system is almost correct, but not quite. It may happen that a customer makes two orders. Assume that, individually, the values of both orders equal the customer's credit limit, so that cumulatively, they exceed it. Therefore, the first order will be accepted, but the second will be rejected. This behaviour will usually be correct. Suppose, however, that the first order concerns a product that is out of stock. Then it too will be rejected, but for a different reason. The customer's balance will have been debited by 'Update Balances', so a compensating credit has to be applied by 'Adjust Balances'. At the end of this process the customer's balance will be the same as its initial value, which is correct, given that both orders were rejected. Unfortunately, the result is still incorrect. If the first order is rejected, there is no reason to reject the second one. The system has lost the opportunity to sell goods to a customer in good credit standing because some other product was out of stock. It may be that this is an acceptable result. Statistically, the likelihood of these, or similar, circumstances occurring in practice is so remote that little revenue would be lost, and the client may be satisfied with the proposed design.

We may call this type of system an 'optimistic fail-safe' design. It is optimistic because it updates the customer balances assuming that there will be no stock shortage. That is, it assumes that some future condition will be satisfied. It is fail-safe because, even if the condition is not satisfied, it cannot lead to a dangerous condition in which customers exceed their credit limits.

It is possible to consider an alternative solution that optimistically updates 'Stock' rather than 'Balance', as shown in Figure 6.7.3. This design would have to assume that no orders would be rejected by exceeding customer credit limits. The stock would be decreased in the 'Update Stock' process, but, if an order was rejected for lack of credit, it might need to be increased again in the 'Adjust Stock' process. Its fault is that, if a product was in short supply, it might allocate its remaining items to customers who could not afford them rather than ones who could. At the end of the process, there could be stock left over, even though there were customers who able to pay for it. Whether this is a better design than that of Figure 6.7.2 depends on the client's business policies.

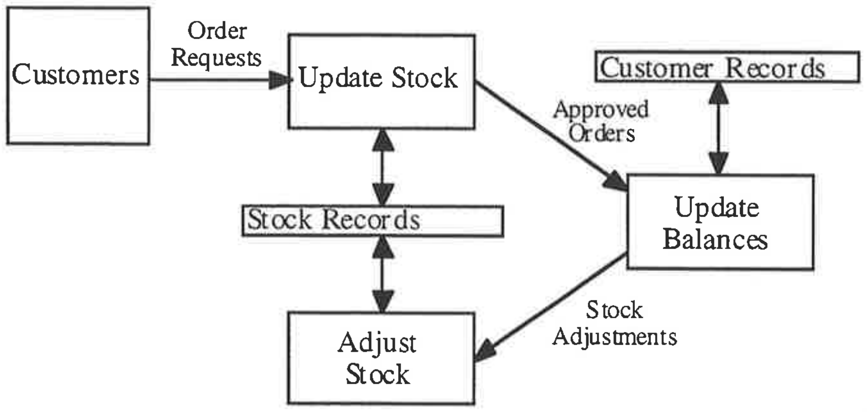

FIGURE 6.7.3: OPTIMISTIC UPDATING OF 'STOCK'

Neither the design of Figure 6.7.2 or of Figure 6.7.3 is real-time equivalent. The design of Figure 6.7.2 introduces unwanted interactions between products, whereas that of Figure 6.7.3 introduces unwanted interactions between customers. As a result, it is not possible to specify either system using the specification language of Section 2.6, at least, not directly. Example 6.7.1 shows how this might be done indirectly. There are two copies of 'Balance': 'Rough_Balance' and 'Exact_Balance'. 'Rough_Balance' is the attribute updated by 'Update Balances' in Figure 6.7.2, whereas 'Exact_Balance' is the attribute determined by 'Adjust Balances'. Analysis of this specification could lead to the design of Figure 6.7.2, because 'Rough_Balance' does not depend on 'Stock', and 'Stock' does not depend on 'Exact_Balance'. However, we have to assume that some other event forces 'Rough_Balance' to agree with 'Exact_Balance' at the start of each batch, by copying all values from 'Exact_Balance' to 'Rough_Balance'. This event would have to be allocated to a separate mode, run *between* batches of orders.

```
with Error, Invoice;
generic
   type customer is private;
   type product is private;
package Order_Processing is
   subtype money is integer;
   procedure Order (Who : customer; What : product; Qty_Ordered : positive);
   -- other event specifications
package body Order_Processing is
   Authorised : array (customer) of boolean := (others => false);
   Rough_Balance, Exact_Balance, Credit_Limit :
               array (customer) of money := (others => 0);
   Offered : array (product) of boolean := (others => false);
   Price : array (product) of money := (others => 0);
   Stock : array (product) of natural := (others => 0);
   procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
      Qty_Delivered, Shortage : natural := 0;
      Value_Ordered, Available_Credit : money := 0;
   begin
      if not Offered (What) then
         Error.Product (What);
      elsif not Authorised (Who) then
         Error.Customer (Who);
      else
         Value_Ordered := Price (What) * Qty_Ordered;
         Available_Credit := Credit_Limit (Who) – Rough_Balance (Who);
         Rough_Balance (Who) := Rough_Balance (Who) + Value_Ordered;
         Exact_Balance (Who) := Exact_Balance (Who) + Value_Ordered;
         if Value_Ordered > Available_Credit then
            Error.Credit (Who, What, Value_Ordered, Available_Credit);
            Exact_Balance (Who) := Exact_Balance (Who) – Value_Ordered;
         else
            Qty_Delivered := min (Stock (What), Qty_Ordered);
            Shortage := Qty_Ordered – Qty_Delivered;
            Invoice.Deliver (Who, What, Qty_Delivered);
            Stock (What) := Stock (What) – Qty_Delivered;
            Exact_Balance (Who) := Exact_Balance (Who) – Shortage * Price (What);
         end if;
      end if;
   end Order;
   -- other event procedures
end Order_Processing;
```

EXAMPLE 6.7.1: SPECIFYING AN OPTIMISTIC FAIL-SAFE 'ORDER'

In summary, it was possible to design an optimistic fail-safe system that was not real-time equivalent to the specification of Example 6.5.1, but which nonetheless may have behaviour acceptable to the client. But this system proved to be real-time equivalent to the alternative specification of Example 6.7.1. This must be so in general. If the system contains more than one process that updates the same attribute, it is merely necessary to make multiple copies of the attribute, and ensure that the specification updates each one as in the optimistic fail-safe design. We must then assume that some other event brings the various copies of the attribute into step. Thus, provided we specify the system we actually want, rather than one that we would like to have, dependence analysis will still lead to the desired design. We make one proviso: the implementation being discussed must be a batch system. If the data flow diagram of Figure 6.7.2 represented a set of concurrent processes linked by queues, there would be less risk of unwanted interactions between events. There would be a good chance that the updates to 'Balance' made by 'Adjust Balances' would occur so soon after the inspections made by 'Update Balances' that the two processes would rarely be out of step. They might become out of step only when an order directly followed another for the same customer. Example 6.7.1 does not describe this situation well.

6.8 Error Detection and Correction

What happens if an event is in error? For example, an order might be placed by an unauthorised customer, or made for a product that is not offered. In practice, this typically means that an error had been made in recording a customer number or a product code. As a result, a customer might fail to get the goods they ordered. One way to deal with this would be to correct the event, and submit it again in the next batch of input. This might be expedient, but it is not logically correct. It might prove that, although sufficient goods were in stock at the time the customer's order *should* have been processed, they were no longer available when it was actually processed. This might be satisfactory in practice, on the grounds that the customer's order might just as easily been delayed in the postal system. However, it is clear that such an approach will not work in general [Inglis 1981, Dwyer 1981b].

The problem with invalid events is that they cause one or more state variables to fail to reach their correct states, and their histories remain incorrect from that time onward. It is therefore necessary not just to resubmit the erroneous event, but all those following it that affect the same state variables. In principle, it would be possible to freeze each state variable in its most recent valid state, and resubmit only those events that are affected by the invalid ones. But there is typically little to be gained in doing this. Using sequential updating, the cost of updating the master files depends only secondarily on the number of events in the batch, so it is almost as cheap, and certainly a lot simpler, to resubmit all the events, starting with the database in its initial state. The same argument does not apply to parallel access, but resubmitting the batch from scratch is still attractive, because of the extra processing needed to do anything more sophisticated than to start again from the previous back-up of the database.

Given that errors in a batch will be corrected by resubmitting it, a more liberal approach may be taken to algorithm design. In the order processing system discussed in the previous section, most actions are conditional on the value of 'Authorised' or 'Offered'. As a simple example, consider the specification in Example 6.8.1 of the 'Close' event from Example 6.4.1. Because the assignment to 'Closed' is conditional, this specification creates a dependence of 'Closed' on 'Authorised'.

```
procedure Close (Who : customer) is
begin
   if not Authorised (Who) then
      Error.Customer (Who);
   else
      Closed (Who) := true;
   end if;
end Close;
```

EXAMPLE 6.8.1: CONDITIONAL SPECIFICATION OF THE 'CLOSE' EVENT.

However, if errors are corrected by resubmitting batches of events, the specification could just as well be written as in Example 6.8.2, in which 'Closed' is independent of 'Authorised'. The point is, that although 'Closed (Who)' is wrongly updated if 'Authorised (Who)' is false, whenever this happens, the batch will be resubmitted anyway. In general, removing dependences is a good thing, because it may remove a cycle from the SDG, perhaps making an efficient solution possible where none was possible before.

```
procedure Close (Who : customer) is
begin
   if not Authorised (Who) then
      Error.Customer (Who);
   end if;
   Closed (Who) := true;
end Close;
```

EXAMPLE 6.8.2: UNCONDITIONAL SPECIFICATION OF THE 'CLOSE' EVENT.

A completely different approach to error detection is to adopt a front-end/back-end design. In such a design, potential errors are eliminated before events are submitted for batch processing.

The simplest kind of front-end process is a 'validation program', which attempts to find errors by statistical means. For example, it can check quantities and values to see if they fall inside the usual ranges. When a value falls outside the normal range, it is not necessarily wrong, but it certainly deserves extra scrutiny. In addition, redundant data may be supplied. For example, customer numbers and product codes may incorporate check digits, so that any transcription errors introduced when copying them will almost certainly be detected. Another use of redundancy is to use 'control totals' calculated by independent means: e.g., the total number of items ordered in the batch can be compared with the sum of the number of items in each event. It is also possible to use redundancy within an event: e.g., the number of items, price, and the value sold may all be provided as data, then a check made that the value sold equals the quantity times the price. These statistical techniques cannot eliminate the possibility that errors will occur in batch processing, but they can greatly reduce the frequency with which batches of events have to be resubmitted.

To ensure that every batch is processed without error — or at least without a detectable error, it is necessary for the front end to check all possible error conditions against the database. Figure 6.8.1 shows one possible way in which the design of Figure 6.6.4 could be adapted. The first process in Figure 6.8.1 represents an interactive program that randomly accesses those database attributes that are used to determine if there are any errors. The remaining three processes constitute the batch processing back end.

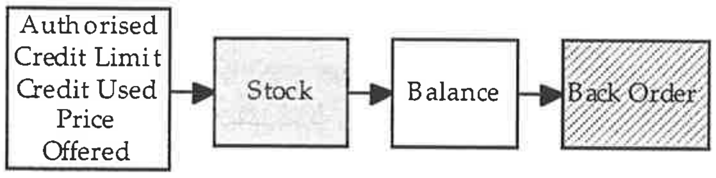

FIGURE 6.8.1: A FRONT-END/BACK-END DESIGN

It might seem that there is little advantage in such a design because, since the front end process must already access the product and customer information, it might as well update 'Stock' and 'Balance' too. However, this overlooks the fact that such a system will almost certainly need to produce reports that involve the whole of the customer and product files. This being so, then the previous argument can be reversed; if it is necessary to access the whole of these files to produce reports, why not update them at the same time? Independently of this argument, it is also the case that the front end design minimises the need to write to the database. Typically, writing one record sequentially as a delayed procedure call to be processed by the back-end process can replace several random-access writes to individual files. Thus, the front-end/back-end design is likely to allow quicker response and greater throughput than a design that processes each event to completion in the front end.

The argument for a front-end/back-end design is even stronger if the front-end operates in read-only mode. The system of Figure 6.8.1 does not permit this, because 'Credit Used' has to be updated. However, is it an error for a customer to exceed their credit limit, or merely an aspect of normal operation? In other words, would the event need to be corrected and resubmitted? If exceeding a credit limit is not an error in this sense, then the front-end could be restricted to checking 'Authorised' and 'Offered'. If it is further assumed that 'Order' events will not be mixed in the same batch with events that can change the 'Authorised' or 'Offered' attributes, then the front end has only to read them. This means that several operators could enter orders concurrently, without the possibility of interaction or the attendant need to lock records. Indeed, it might be possible for the front-end process to read the lists of authorised customers and offered products into main memory once, during initialisation. This would eliminate further accesses to the files by the front end, allowing the maximum possible event throughput and the fastest possible response.

6.9 The Need for Automated Design

The sequence of examples examined in this chapter has been chosen partly to dispel the idea that a efficient and robust system can be designed top-down. It would be wrong to assume that the basic design of a system can be independent of the details of the algorithms involved. An experienced designer may appear to sketch a system design without a detailed knowledge of its procedures, but presumably the designer really knows enough about them to be able to mentally construct the SDG. Moreover, we have seen that a design may need to change dramatically in response to changing requirements — which may prove very costly. The only safe design is a single random-access process, but it is often an order of magnitude less efficient than a less robust independent access design.

A better approach would be to start with a formal specification of the system, derive its SDG, choose an optimum set of composite processes, and transform the system specification into a set of process specifications — all these stages being performed automatically by a system generator CASE tool. (For several examples of existing CASE tools, see [Chikofsky, 1989]). This would dramatically reduce the cost of implementing batch information systems. Assuming that the specifications are stated formally, the CASE tool could have the structure of Figure 6.9.1.

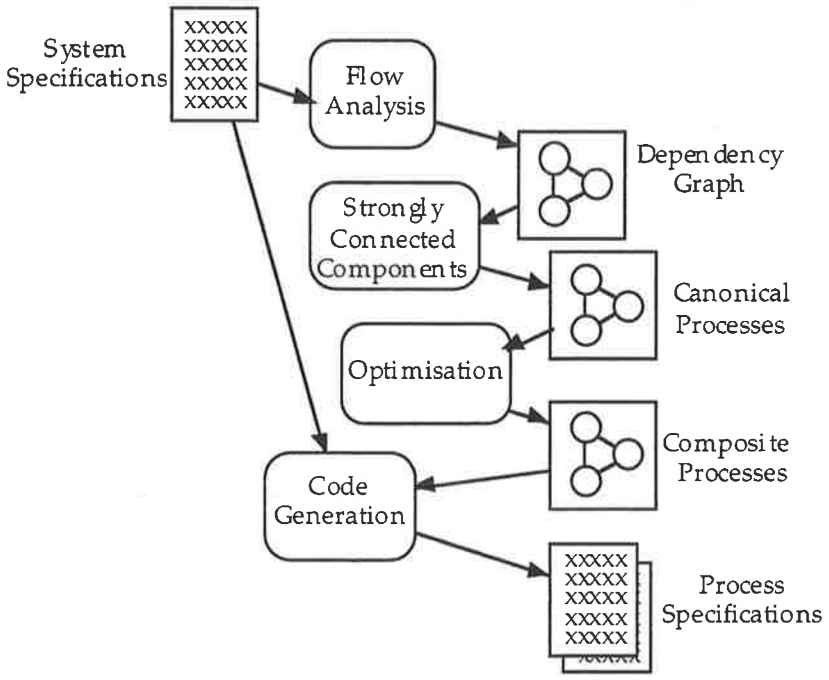

FIGURE 6.9.1: STRUCTURE OF A POSSIBLE CASE TOOL

Starting with a formal specification of the functional dependencies and each kind of event, the flow analysis module extracts the SDG. Next, the strong components of the graph are found, resulting in a CPG. Then an optimiser clusters them into composite processes to improve system performance. Once the set of system components and the database attributes that they may access are known, this knowledge can be used to transform the original specification, resulting in a set of process specifications. If the system specification is procedural, the process specifications can be a set of working programs.

In practice, Figure 6.9.1 would be better structured as two separate CASE tools: a *Designer*, and a *Programmer*. A systems analyst could use the *Designer* tool to explore or confirm the consequences of choosing different data representations and event specifications. However, the optimisation step might not produce the most practical design; there may be some factors in the operating environment that it might fail to take into account. It might be better for the designer to be able to over-ride the result of the automated design stage. The *Programmer* tool needs only to know how attributes are allocated to processes to generate the process specifications.

One reason why a systems analyst might wish to over-ride a design is that external requirements might demand the system to be interactive. As discussed in the preceding section, a system can be divided into an interactive 'front end' that checks for errors, and a batch processing 'back end' that carries out the bulk of the processing. Such environmental requirements are not expressible in the specification language described in Chapter 2, although there is no reason in principle why a specification language should not allow them to be expressed.

Although the *Designer* tool logically precedes the *Programmer*, we discuss the *Programmer* in Chapter 7, and deal with the much more complex *Designer* tool after that. The *Programmer* has not been implemented in software. It would have the usual characteristics of a program generator: it would be straightforward but tedious to write. On the other hand, a prototype *Designer* tool has been built, because the design process is potentially computationally complex. Therefore it is important to demonstrate that design can be completed quickly enough in practice.

7. System Generation

We assume that a *Designer* CASE tool has assigned each variable to a process, or more accurately, allocated each lexical appearance of a variable to a process. In a specification, a local variable could be reused in two unrelated contexts, and the *Designer* might allocate its different appearances to different processes.

As a result of these allocations, each statement of the specification can also be allocated to a process. Often it will be the case that a control statement encloses some assignment statements that become allocated to processes that follow the process allocated to the control statement. But a statement can never become allocated to an earlier process than a statement that encloses it, because the way dependence is defined ensures that it can't happen. An enclosing statement can always activate the statements it encloses, either directly or by delayed procedure call. Therefore, the specifications of component processes can be derived by encapsulating groups of statements as procedures, without a fundamental restructuring of the system specification.

This chapter describes how a system specification can be transformed into a set of process specifications. There are two main stages in this transformation: In the first stage, the specification of each event must be decomposed into a set of event procedures, one or more for each process. In the second, if the process can use independent access, these procedures must be transformed further to fit into the framework of a sequential or parallel update algorithm. There is also a third rather trivial step, which is to embed the event procedures within the update process's read loop.

The reason why we discuss process generation before giving details of the *Designer* tool, is that the requirements of process generation put constraints on the output of the *Designer*, which must be taken into account later. We therefore discuss some of these issues here.

7.1 Event Decomposition

The event decomposition problem is best considered by means of an example. Consider the specification of Example 6.6.2, which discussed an order processing system where customers' orders were subjected to a credit check based on their total credit used. This eventually lead to the pipeline design of Figure 6.6.4, comprising five processes:

1	{Price, Offered}
2	{Authorised, Credit Limit, Credit Used}
3	{Stock}
4	{Balance}
5	{Back Order}

Using this numbering, it is possible to mark the assignments in the specification of the 'Order' procedure with the latest processes in which they can be made, as in Example 7.1.1. Assignments to state variables are simply allocated by their numbering above. Assignments to local variables are allocated by finding the earliest process in which they appear on the right-hand side of an assignment that has already been numbered.

```
    procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
       Qty_Delivered, Shortage : natural := 0;
       Value_Ordered, Available_Credit : money := 0;
    begin
       if not Offered (What) then
          Error.Product (What);
       elsif not Authorised (Who) then
          Error.Customer (Who);
       else
(3)       Value_Ordered := Price (What) * Qty_Ordered;
(3)       Available_Credit := Credit_Limit (Who) – Credit_Used (Who);
          if Value_Ordered > Available_Credit then
             Error.Credit (Who, What, Value_Ordered, Available_Credit);
          else
(3)          Qty_Delivered := min (Stock (What), Qty_Ordered);
(5)          Shortage := Qty_Ordered – Qty_Delivered;
             Invoice.Deliver (Who, What, Qty_Delivered);
(3)          Stock (What) := Stock (What) – Qty_Delivered;
(5)          Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
(4)          Balance (Who) := Balance (Who) + Qty_Delivered * Price (What);
          end if;
(2)       Credit_Used (Who) := Credit_Used (Who) + Value_Ordered;
       end if;
    end Order;
```

EXAMPLE 7.1.1: LATEST PROCESSES FOR ASSIGNMENTS

It is also possible to mark each expression with the earliest process in which it can be fully evaluated, as in Example 7.1.2. As it happens, all the expressions that appear in assignments to system variables in this example can't be evaluated any earlier than in the processes that update the variables, for the simple reason that in each case the assigned variable also appears in the right-hand expression. This need not be true in general, as in the case of the assignment to 'Value Ordered', which can be allocated to any of the first three processes. The situation would be made more complicated if we allowed the possibility of evaluating sub-expressions and assigning them to internal variables. For example, the sub-expression 'Qty_Delivered * Price' in the assignment to 'Balance (Who)' could be evaluated in the 3rd process rather than the 4th.

Consideration of the control structure reveals that although the outermost **if** condition ('**not** Offered (What)') can be evaluated in the 1st process, it need not be used until the 3rd, because all the assignments it encloses can be deferred to the 3rd process. Similarly, the call 'Error.Product (What)' can be made as early as the 1st process, or deferred until the 5th process.

```
    procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
       Qty_Delivered, Shortage : natural := 0;
       Value_Ordered, Available_Credit : money := 0;
    begin
(1)    if not Offered (What) then
(1)       Error.Product (What);
(2)    elsif not Authorised (Who) then
(1)       Error.Customer (Who);
       else
(1)       Value_Ordered := Price (What) * Qty_Ordered;
(2)       Available_Credit := Credit_Limit (Who) – Credit_Used (Who);
(2)       if Value_Ordered > Available_Credit then
(2)          Error.Credit (Who, What, Value_Ordered, Available_Credit);
          else
(3)          Qty_Delivered := min (Stock (What), Qty_Ordered);
(3)          Shortage := Qty_Ordered – Qty_Delivered;
(3)          Invoice.Deliver (Who, What, Qty_Delivered);
(3)          Stock (What) := Stock (What) – Qty_Delivered;
(5)          Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
(4)          Balance (Who) := Balance (Who) + Qty_Delivered * Price (What);
          end if;
(2)       Credit_Used (Who) := Credit_Used (Who) + Value_Ordered;
       end if;
    end Order;
```

EXAMPLE 7.1.2: EARLIEST PROCESSES FOR EXPRESSIONS

Where is the best place to allocate the evaluation of each expression, each assignment, and so on? Compared with choosing the optimum set of processes, this is a minor problem; but it deserves consideration because it determines the number of parameters that have to passed between processes, and therefore the sizes of the messages that are needed to transmit them. The problem is analogous to the optimum allocation of registers in an optimising compiler. Rather than deal with the problem optimally, we suggest some simple heuristics.

The first heuristic is, other things being equal, to evaluate each expression as early as possible. This means that the value of the expression can be passed between processes as a single parameter, whereas the collection of terms that make up the expression would usually need more than one parameter. This argument is not watertight; it may be that many different expressions are composed from a few basic terms, but this is rather unlikely.

Early evaluation also has the dubious advantage of giving some control to the specifier. It means that an expression can be assigned to a local variable in the expectation that it will be evaluated as soon as possible, and therefore the variable will be passed to later processes rather the terms that are used to evaluate it. With this assumption, the specification of Example 6.6.2 would be slightly improved by discarding the local variable 'Value_Ordered', and wherever it is used, using the expression 'Price (What) * Qty_Ordered' instead — the point being that 'Price (What)' and 'Qty_Ordered' have to be passed as parameters to later processes anyway, so 'Value_Ordered' is redundant.

As a corollary of choosing early evaluation, the calls to external packages can also be made as early as possible. Since these external packages often model reporting processes, this makes the system more interactive. This is particularly valuable if the outputs are error reports, as it may be sensible to abandon processing and correct the errors before continuing. If the outputs are destined become the inputs to other systems, producing them early means that these systems are not held up waiting for data.

Early evaluation also makes sense for **if** statements. For example, if the outermost **if** statement in Example 7.1.1 and Example 7.1.2 succeeds, an external output is produced, but nothing else happens, so no messages will be passed between the processes. In general, placing **if** statements as early as possible can only serve to reduce the number of messages that flow through the system.

If an assignment is nested inside an **if** statement, it may turn out that its right-hand expression could be evaluated in an earlier process than the **if** expression can. This is the case for the assignment to 'Value_Ordered' in Example 7.1.2, whose expression could be evaluated in the first process, but whose controlling condition cannot be evaluated until the second process. In such cases it is important that the *Designer* tool should defer the evaluation of the assignment expression until the process that evaluates the control condition — it may be that the purpose of the **if** statement is to ensure that the expression can be evaluated safely, e.g., by protecting it against a division by zero. (On the other hand, if an expression can safely be evaluated outside an **if** statement, it may pay the specifier to place it there; it might remove a dependence and permit a more efficient implementation to be found.) Clearly calls to external packages must follow the same rule; although the parameter of the call 'Error.Customer (Who)' is known in the first process, the call certainly shouldn't be made until the appropriate conditions have been tested, which must wait until the second process.

In fact, these situations are handled without difficulty, because the 3rd rule in Section 4.5 forces us to regard the control conditions surrounding an assignment or call as if they were part of the right-hand expression itself. With these rules in mind, the assignment to 'Value_Ordered' and the call 'Error.Customer (Who)' would be allocated to the 2nd process rather than the 1st.

The resulting allocation of expressions to processes is shown in Example 7.1.3, which may also be interpreted as defining the placement of the corresponding statements into processes. For example, if a control expression should be evaluated in the 2nd process, then the text of its corresponding **if** statement should also appear in the 2nd process. To make the example a bit clearer, the **elsif** clause in Example 7.1.1 and Example 7.1.2 has been expanded into an **else** clause containing an **if** statement.

```
    procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
       Qty_Delivered, Shortage : natural := 0;
       Value_Ordered, Available_Credit : money := 0;
    begin
(1)    if not Offered (What) then
(1)       Error.Product (What);
       else
(2)       if not Authorised (Who) then
(2)          Error.Customer (Who);
          else
(2)          Value_Ordered := Price (What) * Qty_Ordered;
(2)          Available_Credit := Credit_Limit (Who) – Credit_Used (Who);
(2)          if Value_Ordered > Available_Credit then
(2)             Error.Credit (Who, What, Value_Ordered, Available_Credit);
             else
(3)             Qty_Delivered := min (Stock (What), Qty_Ordered);
(3)             Shortage := Qty_Ordered – Qty_Delivered;
(3)             Invoice.Deliver (Who, What, Qty_Delivered);
(3)             Stock (What) := Stock (What) – Qty_Delivered;
(5)             Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
(4)             Balance (Who) := Balance (Who) + Qty_Delivered * Price (What);
             end if;
          end if;
(2)       Credit_Used (Who) := Credit_Used (Who) + Value_Ordered;
       end if;
    end Order;
```

EXAMPLE 7.1.3: EARLY ASSIGNMENT OF EXPRESSIONS TO PROCESSES

Given such an allocation, generating the texts of the process specifications is straightforward. Considering the 1st process, it should contain all the statements labelled (1) in Example 7.1.3, replacing all the rest by delayed procedures. The first decomposition is shown in Example 7.1.4a and Example 7.1.4b.

Apart from allocating the correct statements to each process, it is also necessary to ensure each procedure has the correct parameters. In general, processes can only refer to variables allocated to them, or variables that are accessed in earlier processes that are passed to them as input parameters. Since the first process accesses only 'Offered (What)', which is not referred to in the 2nd process, it is not needed as a parameter.

```
    procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
    begin
(1)    if not Offered (What) then
(1)       Error.Product (What);
       else
(2)       Process_2.Order (Who, What, Qty_Ordered);
       end if;
    end Order;
```

EXAMPLE 7.1.4A: THE 1ST EVENT PROCEDURE

```
        procedure Order (Who : customer; What : product;
                              Qty_Ordered : positive; Price : money) is
           Qty_Delivered, Shortage : natural := 0;
           Value_Ordered, Available_Credit : money := 0;
        begin
(2)        if not Authorised (Who) then
(2)           Error.Customer (Who);
           else
(2)           Value_Ordered := Price * Qty_Ordered;
(2)           Available_Credit := Credit_Limit (Who) – Credit_Used (Who);
(2)           if Value_Ordered > Available_Credit then
(2)              Error.Credit (Who, What, Value_Ordered, Available_Credit);
              else
(3)              Qty_Delivered := min (Stock (What), Qty_Ordered);
(3)              Shortage := Qty_Ordered – Qty_Delivered;
(3)              Invoice.Deliver (Who, What, Qty_Delivered);
(3)              Stock (What) := Stock (What) – Qty_Delivered;
(5)              Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
(4)              Balance (Who) := Balance (Who) + Qty_Delivered * Price;
              end if;
(2)           Credit_Used (Who) := Credit_Used (Who) + Value_Ordered;
           end if;
        end Order;
```

EXAMPLE 7.1.4B: THE 2ND TO 5TH EVENT PROCEDURES

```
        procedure Order (Who : customer; What : product;
                              Qty_Ordered : positive; Price : money) is
           Value_Ordered, Available_Credit : money := 0;
        begin
(2)        if not Authorised (Who) then
(2)           Error.Customer (Who);
           else
(2)           Value_Ordered := Price * Qty_Ordered;
(2)           Available_Credit := Credit_Limit (Who) – Credit_Used (Who);
(2)           if Value_Ordered > Available_Credit then
(2)              Error.Credit (Who, What, Value_Ordered, Available_Credit);
              else
(3)              Process_3.Order (Who, What, Qty_Ordered, Price);
              end if;
(2)           Credit_Used (Who) := Credit_Used (Who) + Value_Ordered;
           end if;
        end Order;
```

EXAMPLE 7.1.5A: THE 2ND EVENT PROCEDURE

The factorisation of the second process proceeds in a similar way. Except for the innermost **else** clause, the whole text can be allocated to the 2nd process. This gives the decomposition shown in Example 7.1.5a and Example 7.1.5b. Inspection of the text of the 3rd and following processes reveals that 'Price (What)' needs to passed to them as an extra parameter. It must declared as a formal parameter, 'Price', in the text of the 3rd and following processes and then substituted for 'Price (What)' throughout. The declaration of the 'Price' formal parameter is derived straightforwardly from the type of the 'Price' state variable. Because 'Value_Ordered' and ‘Available_Credit’ are evaluated in the 2nd process, it is also necessary to declare them there.

```
     procedure Order (Who : customer; What : product;
                         Qty_Ordered : positive; Price : money) is
        Qty_Delivered, Shortage : natural := 0;
     begin
(3)     Qty_Delivered := min (Stock (What), Qty_Ordered);
(3)     Shortage := Qty_Ordered – Qty_Delivered;
(3)     Invoice.Deliver (Who, What, Qty_Delivered);
(3)     Stock (What) := Stock (What) – Qty_Delivered;
(5)     Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
(4)     Balance (Who) := Balance (Who) + Qty_Delivered * Price;
     end Order;
```

EXAMPLE 7.1.5B: THE 3RD TO 5TH EVENT PROCEDURES

Continuing the decomposition, the 3rd process must be decomposed as in Example 7.1.6a and Example 7.1.6b. ‘Qty_Delivered’ and ‘Shortage’ have to be declared in the 3rd process and passed as parameters to the 4th process.

```
     procedure Order (Who : customer; What : product;
                         Qty_Ordered : positive; Price : money) is
        Qty_Delivered, Shortage : natural := 0;
     begin
(3)     Qty_Delivered := min (Stock (What), Qty_Ordered);
(3)     Shortage := Qty_Ordered – Qty_Delivered;
(3)     Invoice.Deliver (Who, What, Qty_Delivered);
(3)     Stock (What) := Stock (What) – Qty_Delivered;
(4)     Process_4.Order (Who, What, Price, Qty_Delivered, Shortage);
     end Order;
```

EXAMPLE 7.1.6A: THE 3RD EVENT PROCEDURE

```
     procedure Order (Who : customer; What : product; Price : money;
                         Qty_Delivered, Shortage : natural) is
     begin
(5)     Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
(4)     Balance (Who) := Balance (Who) + Qty_Delivered * Price;
     end Order;
```

EXAMPLE 7.1.6B: THE 4TH AND 5TH EVENT PROCEDURES

The final decomposition is trivial, and is shown in Example 7.1.7a and Example 7.1.7b.

```
     procedure Order (Who : customer; What : product; Price : money;
                         Qty_Delivered, Shortage : natural) is
     begin
(5)     Process_5.Order (Who, What, Shortage);
(4)     Balance (Who) := Balance (Who) + Qty_Delivered * Price;
     end Order;
```

EXAMPLE 7.1.7A: THE 4TH EVENT PROCEDURE

```
    procedure Order (Who : customer; What : product; Shortage : natural) is
    begin
(5)   Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
    end Order;
```

EXAMPLE 7.1.7B: THE 5TH EVENT PROCEDURE

It may seem almost miraculous that the decomposition proceeds so smoothly. However, there should be no surprise; the allocation of variables to processes was made to ensure that it would do so. Likewise, the definition of dependence made in Section 4.5 was chosen with the same goal in mind.

The decomposition into the 3rd, 4th and 5th processes is somewhat arbitrary. The decomposition given above follows the process pipeline design of Figure 6.6.4. Because the composite process graph of Figure 6.6.3 forces no ordering on 'Back_Order' and 'Balance', the order of the last two steps could be exchanged. Better still they could be independent; the 'Stock' update could send delayed calls to each one directly. The system design would no longer be a simple pipeline, but would branch. However, since the advantage claimed for a pipeline design is that it does not have to merge messages from different sources, the branching would introduce no additional technical difficulty. (For that matter, if a system is generated automatically by a computer, a pipeline design has no advantage; the technical problem of merging data-flows has only to be solved once, then incorporated into the *Programmer* tool.)

It is worth making one final comment. Suppose the assignments in the innermost **else** clause had been specified in a different order, so that Example 7.1.5b had read like Example 7.1.8a, separating the assignments to 'Back_Order' and 'Balance'. Assuming again that a pipeline is used, and messages are not passed directly from the 3rd process to the 5th process, the obvious decomposition is that of Example 7.1.8b, which uses two delayed calls instead of the single call used in Example 7.1.6a. In fact, it must always be safe to adjust the order of the assignments within a sequence so that only one call is needed. In fact any delayed call can always be safely placed as the last statement of a sequence. The reason is simple, delayed calls have no output parameters; any assignments made by the procedure invoked by the call cannot affect variables in the calling process.

```
    procedure Order (Who : customer; What : product; Qty_Ordered : positive;
                     Price : money) is
      Qty_Delivered, Shortage : natural := 0;
    begin
(3)   Qty_Delivered := min (Stock (What), Qty_Ordered);
(3)   Shortage := Qty_Ordered – Qty_Delivered;
(5)   Back_Order (Who, What) := Back_Order (Who, What) + Shortage;
(3)   Invoice.Deliver (Who, What, Qty_Delivered);
(3)   Stock (What) := Stock (What) – Qty_Delivered;
(4)   Balance (Who) := Balance (Who) + Qty_Delivered * Price;
    end Order;
```

EXAMPLE 7.1.8A: A MODIFIED ORDER OF ASSIGNMENTS

```
    procedure Order (Who : customer; What : product; Qty_Ordered : positive;
                         Price : money) is
        Qty_Delivered, Shortage : natural := 0;
    begin
(3)     Qty_Delivered := min (Stock (What), Qty_Ordered);
(3)     Shortage := Qty_Ordered – Qty_Delivered;
(5)     Process_4.Order_1 (Who, What, Shortage);
(3)     Invoice.Deliver (Who, What, Qty_Delivered);
(3)     Stock (What) := Stock (What) – Qty_Delivered;
(4)     Process_4.Order_2 (Who, Qty_Delivered, Price);
    end Order;
```

EXAMPLE 7.1.8B: A MODIFIED VERSION OF THE 3RD PROCESS

Only one thing spoils the simple early evaluation approach: loops should be placed as far *downstream* as possible. The argument is the converse of that for **if** statements; it is better to pass one message to activate a whole loop than to pass many separate messages to activate each of its iterations.

It is easy to deal with **while** loops. Any sensible **while** loop must contain at least one assignment that modifies the value of its control condition, otherwise it has no means of terminating. This effectively binds the loop to the same process as the assignment, ensuring that at least part of the loop body is allocated to the same process as the loop itself. It does not exclude the possibility that the loop body will need to make calls to a later process, because it may also include an assignment to a variable allocated to the later process. However, optimising the process graph will tend to group as many assignments with the loop body as possible, provided they are compatible with it. If they are incompatible, multiple calls are unavoidable.

The treatment of **all** and **for** loops is a little harder. The reason is due to a defect of the specification language. It is possible to write '**all** i **in** index **loop** ...', suggesting that all values of 'i' in the domain 'index' should be generated, whereas the intention is that only 'interesting' values should be generated, specifically those for which at least one attribute accessed in the loop body has a non-default value. For example the specification might read:

```
all p in product loop
    if Offered (p) then
        ...
    elsif Stock (p) > 0 then
        ...
    end if;
end if;
```

where it is implicit that only those values of 'p' that have 'Offered' equal to true, or a non-zero value of 'Stock' should be examined. This assumption could be made explicit by changing the syntax of an **all** or **for** loop as follows:

```
all p in product with Offered or Stock loop
   if Offered (p) then
      ...
   elsif Stock (p) > 0 then
      ...
   end if;
end if;
```

This makes it clear that the loop should be placed in whatever process inspects ‘Offered’ and ‘Stock’. In practice, **all** and **for** loops cause little difficulty. Essentially, there is never any reason to separate loop variables from loop bodies. Provided it places loops as far downstream as possible, the optimisation process ensures that they gravitate together. There is nothing to keep them apart.

7.2 Transformation to Independent Access

```
procedure Order (Who : customer; What : product;
                    Price : money; Qty_Ordered : positive) is
   Value_Ordered, Available_Credit : money := 0;
begin
   all c in customer loop
      if c = Who then
         if not Authorised (c) then
            Error.Customer (c);
         else
            Value_Ordered := Price * Qty_Ordered;
            Available_Credit := Credit_Limit (c) – Credit_Used (c);
            if Value_Ordered > Available_Credit then
               Error.Credit (Who, What, Value_Ordered, Available_Credit);
            else
               Process_3.Order (Who, What, Qty_Ordered, Price);
            end if;
            Credit_Used (c) := Credit_Used (c) + Value_Ordered;
         end if;
      end if;
   end loop;
end Order;
```

EXAMPLE 7.2.1: THE 2ND EVENT PROCEDURE ADAPTED FOR INDEPENDENT ACCESS

The second stage of transformation, which applies only to independent access processes, is to embed each event procedure into the structure of a parallel or sequential update algorithm. This can be envisaged as occurring in two stages: first, each event is transformed into a independent access process, and second, the operations on the events are interleaved. Suppose that a process allows independent access by ‘customer’. Then the first stage requires each event procedure to include an **all** or **for** loop whose loop variable has the domain ‘customer’. In the case of a composite key, such as ‘(customer, product)’ nested loops are needed. Access can only be made to elements that are indexed by the loop variable. Access to arbitrary elements is achieved by selecting the iteration where the loop variable takes the desired value.

For example, the event specification of Example 7.1.5a would be transformed for independent access by customer as in Example 7.2.1.

Although this transformation is very simple, it captures the essence of both the sequential and parallel update algorithms. They differ mainly in the means used to exploit the **all** loop, as explained in Chapter 3. A secondary difference is that they use different means to reduce the storage needed to process events. The sequential update sorts the events sequentially so that they can be matched systematically against the records of the master file. The parallel update sorts the events spatially, so that a parallel processor receives only those events that are relevant to it.

What about more difficult events, in which more than one element is inspected? Recall the 'Careful' transaction between two accounts of Example 5.2.5. This can be factorised into two processes as shown in Example 7.2.2a and Example 7.2.2b.

```
procedure Careful_Transfer (From, To: account; Amount : money) is
begin
   if Authorised(From) and Authorised(To) then
      Accounting_2.Careful_Transfer (From, To, Amount);
   end if;
end Careful_Transfer;
```

EXAMPLE 7.2.2A: THE 1ST EVENT PROCEDURE FOR A 'CAREFUL' TRANSACTION

```
procedure Careful_Transfer (From, To: account; Amount : money) is
begin
   Balance(From) := Balance(From) – Amount;
   Balance(To) := Balance(To) + Amount;
end Careful_Transfer;
```

EXAMPLE 7.2.2B: THE 2ND EVENT PROCEDURE FOR A 'CAREFUL' TRANSACTION

The second procedure is the easier one to transform. It is just an extension of previous principles, as shown in Example 7.2.3.

```
procedure Careful_Transfer (From, To: account; Amount : money) is
begin
   all a in account loop
      if a = From then
         Balance(a) := Balance(a) – Amount;
      end if;
      if a = To then
         Balance(a) := Balance(a) + Amount;
      end if;
   end loop;
end Careful_Transfer;
```

EXAMPLE 7.2.3: THE 2ND EVENT PROCEDURE TRANSFORMED FOR INDEPENDENT ACCESS

The first procedure is a bit trickier, as explained in Section 3.1, it requires both elements of 'Authorised' to be brought together in order to evaluate the control expression. This must be done in the 'collection' phase of the update, as shown in Example 7.2.4.

```
procedure Careful_Transfer (From, To: account; Amount : money) is
begin
   all a in account loop
      if a = From then
         Authorised_From := Authorised(a);
      end if;
      if a = To then
         Authorised_To := Authorised(a);
      end if;
   end loop;
   if Authorised_From and Authorised_To then
      Accounting_2.Careful_Transfer (From, To, Amount);
   end if;
end Careful_Transfer;
```

EXAMPLE 7.2.4: THE 1ST EVENT PROCEDURE TRANSFORMED FOR INDEPENDENT ACCESS

In Section 2.5, it was explained how arrays of packages could be used to model independent access, so, logically, a specification such as that of Example 7.2.2a should be transformed again to exploit arrays of processes. However, process arrays are irrelevant to the actual task of programming, because the final steps of the transformation have to be tailored to generate parallel or sequential updates. The important point is that if a specification can be put into a form similar to that of Example 7.2.2a, then it must be possible to incorporate it into an update algorithm.

The above examples concern cases where the original event specification does not contain a loop, so a loop must be added in order to model independent access. The alternative is that the specification already contains an **all** loop, so no transformation is needed. However, it is clear that the domains of the loops must agree; it would not be sensible to access all customers in a process that updates all products.

From the foregoing, it will be seen that a procedure can always be transformed into a form suitable for independent access provided the frequency with which its statements are executed is related to the degree of independence of the **all** loop. If their frequency is the same as the degree of independence, the event is broadcast to each instance of the loop body; if it is less, the event is sent to selected instances. This result generalises to the case of composite domains. For example, if an attribute is indexed by '(customer, product)' and allows independent access, then an event procedure may contain nested loops, such as '**all** c **in** customer **loop** **all** p **in** product **loop** ...', a single loop, such as '**all** c **in** customer **loop** ...' or '**all** p **in** product **loop** ...', or no loop at all.

7.3 Loop Structures

Although the treatment of **all** loops is straightforward, there are many other interesting cases to consider, e.g., specifications that contain **while** loops, nested loops, or sequences of loops. Only certain structures can be embedded into independent access processes. Therefore, it is important for the *Designer* tool to be able to distinguish these structures, so that the

Programmer's task is feasible. It is pointless for the *Designer* to decide, on the basis of dependence analysis, that independent access is possible, unless the structure of the algorithm has a realisable loop structure.

We may distinguish two kinds of **while** loop. The first is illustrated by the following example:

```
Order_Qty (What) := 0;
while Order_Qty (What) < Shortage (What) loop
   Order_Qty (What) := Order_Qty (What) + Economic_Order_Qty (What);
end loop;
```

EXAMPLE 7.3.1: A 'COMPATIBLE' WHILE LOOP

which sets 'Order_Qty (What)' to the least multiple of 'Economic_Order_Qty (What)' that is no less than 'Shortage (What)'. The loop in Example 7.3.1 has little effect, because it could be replaced by:

```
Order_Qty(What) := least_multiple (Shortage(What), Economic_Order_Qty(What));
```

where 'least_multiple' is a function. The loop involves only one value of 'What', so there can be no interaction between different array elements.

The second kind of **while** loop is illustrated by:

```
while Boss (Who) /= null loop
   Who := Boss (Who);
end loop;
```

EXAMPLE 7.3.2: AN 'INCOMPATIBLE' WHILE LOOP

The loop in Example 7.3.2 cannot be replaced by a function, unless we write:

```
Who := highest_rank(Who, Boss);
```

where the whole 'Boss' array would have be passed as an argument to the function, not just one element of it. In particular, the loop involves multiple values of 'Boss (Who)', and furthermore, each value depends on the previous one. It should be clear that this algorithm cannot be implemented using independent access. The two cases are easily distinguished, because the 'compatible' **while** loop does not assign new values to an index, whereas the 'incompatible' **while** loop does. For the distinction to be apparent from an SDG, 'Boss', must be marked as incompatible with itself as in Figure 7.3.1. (The argument is similar to that in Section 5.6.)

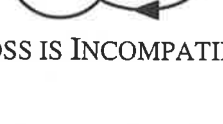

FIGURE 7.3.1: BOSS IS INCOMPATIBLE WITH ITSELF

The rules for constructing dependences ensure that the nesting order of loops and conditionals can be preserved when generating process specifications. An absence of incompatible edges also determines when independent access is possible in principle. However, since dependences suppress the syntactical structure of the specification, they cannot guarantee that the loop structures assigned to a process are consistent with the parallel or sequential update algorithms. That is to say, there is a danger that dependence analysis may discover that independent access is possible, but it achieving it may involve a deeper transformation of the specification that can be achieved by the means discussed above.

Before considering cases involving unrealisable control structures, it is instructive to see how a cartesian product may be formed, as in the specification of Example 7.3.3.

```
package body Cartesian is
   A : array (domain_A) of type_A;
   B : array (domain_B) of type_B;
   procedure product is
   begin
      all i in index loop
         all j in domain loop
            report.product (A(i), B(j));
         end loop;
      end loop;
   end product;
end Cartesian ;
```

EXAMPLE 7.3.3: FORMING A CARTESIAN PRODUCT

The analysis will factorise the specification into an 'A' process and a 'B' process as in Example 7.3.4.

```
package body Cartesian_A is
   A : array (domain_A) of type_A;
   procedure product is
   begin
      all i in index loop
         Cartesian_B.product (A(i));
      end loop;
   end product;
end Cartesian_A;
package body Cartesian_B is
   B : array (domain_B) of type_B;
   procedure product (A : type_A) is
   begin
      all j in domain loop
         Report.product (A, B(j));
      end loop;
   end product;
end Cartesian_B;
```

EXAMPLE 7.3.4: CARTESIAN PRODUCT PROCESSES

There is an implementation problem here: 'Cartesian_A.product' will create a call of 'Cartesian_B.product' for each value of 'i'. The 'Cartesian_B' package must store all these calls throughout the inspection of 'B'. Typically, there will be too many calls to store them all in main memory. The solution is to store the calls on secondary storage, and to read them into main memory in batches. Each batch can then be matched to an inspection of ‘B’. This is a general strategy that can be applied to any batch of events that is too big to fit into main storage. (Similar remarks were made concerning the use of the ‘Memo’ array in Section 3.3.) In this case, it also corresponds to the usual method of forming a cartesian product.

Example 7.3.3 shows that an event procedure may be called an unlimited number of times within the execution of a single event. On the other hand, an independent access update has only two internal means of looping: it may use one or more **all** loops to access the elements of an array, and it may include **while** loops. However, the **while** loops cannot enclose the **all** loops, because this would imply a series of update phases rather than one. However, there can be a **while** loop within the processing of each element, and there can be **while** loops in the distribution and collection phases.

Usually, if an **all** loop is nested within a **while** loop, the termination of the **while** loop will depend on the execution of the **all** loop, as in Example 7.3.5.

```
T := 0;
while T < 100 loop
   all i in index loop
      T := T + A(i);
   end loop;
end loop;
```

EXAMPLE 7.3.5: INTERACTING LOOPS

In this case, dependence analysis will determine that the accesses to ‘A’ are incompatible. Because of the assignment, ‘T’ depends on ‘A(i)’. In turn, ‘A(i)’ depends on the **all** loop variable, ‘i’, which in turn depends on ‘T’ because ‘T’ appears in the expression that controls the **while** loop. This links together definitions of ‘A(i)’ for different values of ‘i’, so that independent access to ‘A’ is impossible.

However, it is possible to create rather artificial specifications where the number of iterations of an outer **while** loop is independent of the inner **all** loop. Consider the example of Example 7.3.6, whose effect is to multiply every element of ‘A’ by eight:

```
procedure Times8 is
   Count: natural;
begin
   Count := 3;
   while Count/= 0 loop
      all i in index loop
         A (i) := A (i) + A (i);
      end loop;
      Count := Count – 1;
   end loop;
end Times8;
```

EXAMPLE 7.3.6: NON-INTERACTING LOOPS

It is a property of this procedure that ‘Count’ and ‘A’ are separable, that is, the number of executions of the outer loop does not depend on the number of executions of the inner loop.

This is not a valid structure for independent access, because it has multiple update phases. On the other hand, this is not revealed by dependence analysis, because the actions on each element of 'A' are independent of one another. In effect, dependence analysis establishes that the multiple update phases are not really needed. This can be seen to be true by transforming the specification to that of Example 7.3.7.

```
procedure Times8 is
   Count: array index of natural;
begin
   all i in index loop
      Count (i) := 3;
      while Count(i) /= 0 loop
         A (i) := A (i) + A (i);
      end loop;
      Count (i) := Count (i) – 1;
   end loop;
end Times8;
```

EXAMPLE 7.3.7: NON-INTERACTING LOOPS

Unfortunately, Example 7.3.6 cannot be transformed into Example 7.3.7 using the rewriting rules. Therefore, if the *Designer* tool were to decide that Example 7.3.6 permitted independent access, it would be beyond the capability of the *Programmer* tool to implement it.

However, if the **while** loop and **all** loop are allocated to separate processes, the *Programmer* could use the ordinary rewriting rules to derive both processes. This is an interesting possibility. The first process accesses a local variable ('Count'), but no state variables. The first process will generate three delayed procedure calls, and the second process will deal with each call in turn as it processes each element of 'A', effectively iterating each assignment statement three times. The effect at execution time would be almost as if the specification had been transformed to that of Example 7.3.7.

In cases of this kind, although dependence analysis suggests that the accesses to 'A' are independent, the *Designer* tool must detect that the specification of Example 7.3.6 cannot be implemented as an independent update process because it has the wrong loop structure. The rule is simple: an **all** loop cannot exploit independent access if it is nested within a **while** loop in the same component process.

7.4 Frequency

In order to preserve enough information about syntactical structure to determine whether loops are properly nested, the *Designer* associates each statement with a 'frequency', which is a measure of how often it is executed. The frequency is simply a list of the **all** or **for** loop variables that enclose the statement in the specification — or in the case of a **while** loop, a dummy variable associated with the control condition. Because independent access is modelled by one or more nested **all** loops, only that part of the frequency that consists of nested **all** loops has the correct structure to be embedded in an independent access process. (In the case of sequential access, we may also include **for** loops.) Thus, the specification of Example 7.3.6 cannot be implemented by independent access in one process. However, once the **while** loop and **all** loop are separated, the **all** loop can exploit independent access.

In the specification, the frequency of the assignment in Example 7.3.6 is ('**while** Count/=0', '**all** i **in** index'). After separating the two loops into different processes, the event procedure in the second process is called with frequency ('**while** Count/=0'), so it must have frequency ('**all** i **in** index'). Since the second process is independent with respect to 'index', the **all** loop can be embedded into the access algorithm. The frequency with which a statement is executed in the specification therefore has two factors: its frequency within the process to which it allocated, and the frequency with which the event procedure in that process is called.

In turn, the frequency of a statement within a process breaks down into two further factors: the frequency due to the **all** loops modelling independent access, and its frequency within each independent sub-process. If the leading terms of the frequency match those for modelling independent access, they can be embedded into the access algorithm; those that remain must be executed within the sub-processes. In the case of the assignment in Example 7.3.6, there is no remainder, so the assignment is made once only. However, in the case of the **while** loop in Example 7.3.1, it cannot be embedded, so the whole loop must be executed for each relevant value of 'What'.

An important aspect of the example of Example 7.3.6 is that a **while** loop can never match an **all** loop. Therefore, it is not until the outer **while** loop has been stripped off by the first process that the inner **all** loop can be matched against the **all** loop of an access algorithm.

On the other hand, it is not necessary that the leading terms of the frequency actually match all the loops in the access algorithm. The effects of the loops in the algorithm can be suppressed by selecting only a specific value or values. In effect, to evaluate:

```
A(k) := A(k) + 1;
```

for some specific value of 'k', an independent access process evaluates, in effect:

```
all i in index loop
   if i = k then
      A(i) := A(i) + 1;
   end if;
end loop;
```

7.5 Incompatible Loops

```
with Report;
generic
   type employee is private;
   type money is range <>;
package Personnel is
   procedure Percent;
end Personnel;
package body Personnel is
   Salary : array (employee) of money := (others => 0);
   procedure Percent is
      Total : money := 0;
   begin
      all Emp1 in employee loop
         Total := Total + Salary(Emp1);
      end loop;
      all Emp2 in employee loop
         Report.Percent(Emp2, Salary(Emp2)*100/Total);
      end loop;
   end Percent;
end Personnel;
```

EXAMPLE 7.5.1: FINDING PERCENTAGES OF A TOTAL

Another problem arises if loops are not nested, but are executed one after another. The specification of Example 7.5.1 reports each employee's salary as a percentage of the total of all salaries. There are two loops: the first finds the total, the second reports the percentages. Taken separately, each loop could use independent access. However, the two loops cannot be interleaved, because the correct total is not found until after the first loop has completed, nor can they be placed in separate processes, because they both access 'Salary', and we assume that 'Salary' could be updated by other events. The only way to implement this specification, as it stands, is as a single-thread process.

```
with Report;
generic
   type employee is private;
   type money is range <>;
package Personnel is
   procedure Percent;
end Personnel;
package body Personnel is
   Salary : array (employee) of money := (others => 0);
   procedure Percent is
      Temp : array (employee) of money := 0;
      Total : money := 0;
   begin
      all Emp1 in employee loop
         Temp(Emp1) := Salary(Emp1);
         Total := Total + Salary(Emp1);
      end loop;
      all Emp2 in employee loop
         Report.Percent(Emp2, Temp(Emp2)*100/Total);
      end loop;
   end Percent;
end Personnel;
```

EXAMPLE 7.5.2: USING A LOCAL ARRAY

As a practical matter, the same effect can be achieved more efficiently by copying the 'Salary' array into the local array 'Temp', effectively taking a snapshot of it, as specified in Example 7.5.2. Both loops now access the 'Temp' array, but only the first accesses the 'Salary' array. However, since there is separate, private instance of the 'Temp' array for each 'Percent' event, there is no longer any possibility that it might be changed by other events. Furthermore, the array can be safely passed between processes by reference rather than value. Because the loops on 'Emp1' and 'Emp2' are not hierarchically nested, we must treat them as incompatible. This strongly suggests the loops should be placed in separate processes, making it possible for them both to use independent access.

In contrast, the specification of Example 7.5.3 presents something of a dilemma. It is clear that Example 7.5.3 can be transformed into Example 7.5.4 by folding the two loops into one. So the two loops are not incompatible in the sense that it is possible to interleave them within a single independent access process. The distinction between this example and Example 7.5.3 is that there is no longer a variable appearing in the second loop whose value depends on the completion of the first loop. So a better rule might be to say that two loops are incompatible only when they *both* are improperly nested, *and* have a dependence linking them.

```
with Report;
generic
   type customer is private;
   type money is range <>;
package Accounts is
   procedure Averages;
end Accounts;
package body Accounts is
   Authorised : array (customer) of boolean := (others => false);
   Balance : array (customer) of money := (others => 0);
   procedure Averages is
      Ave, Total : money := 0;
      Count : integer := 0;
   begin
      all Who1 in customer loop
         Total := Total + Balance (Who1);
      end loop;
      all Who2 in customer loop
         if Authorised (Who2) then
            Count := Count + 1;
         end if;
      end loop;
      Ave := Total / Count;
      Report.Balance(Ave);
   end Averages;
end Accounts;
```

EXAMPLE 7.5.3: FINDING AN AVERAGE

The dilemma arises because, although this rule would then permit the efficient solution given in Example 7.5.4, it can't be derived from Example 7.5.3 by using the existing rewriting rules. Rather than extend them, we insist that loops in specifications can be embedded into independent access processes only if they are hierarchically nested — period.

```
procedure Averages is
   Ave, Total : money := 0;
   Count : integer := 0;
begin
   all Who in customer loop
      Total := Total + Balance (Who);
      if Authorised (Who) then
         Count := Count + 1;
      end if;
   end loop;
   Ave := Total / Count;
   Report.Balance(Ave);
end Averages;
```

EXAMPLE 7.5.4: LOOP FOLDING

8. Use-Definition Analysis

This chapter discusses how the dependence analysis discussed in Chapter 4 can be made rigorous enough to be used in a *Designer* CASE tool.

It was suggested earlier that dependence analysis should really be carried out with definitions of variables rather than the variables themselves. This is because the same variable may be reused in unrelated contexts. Analysing the dependences that determine separability needs techniques that are already well known to compiler writers. These methods consider arrays as single objects. However, analysing compatibility requires extension of the usual analysis techniques to allow elements of arrays to be considered as separate objects. Analysing loops requires a further extension, so that the analysis must distinguish 'dynamic' definitions from 'lexical' definitions.

The input to use-definition analysis is an abstract syntax tree derived by parsing a specification. The parsing process is described in Chapter 10. However, its details are of little importance here — on the contrary, the output from the parsing process is determined by the needs of use-definition analysis.

8.1 Introduction

An SDG is a kind of use-definition graph. Its vertices are definitions of variables, and its edges represent 'uses'. A variable is given a definition whenever it is assigned a new value, typically by an assignment statement. A 'use' represents the use of one definition in constructing another. For example, given the assignment 'x:=y;', a use-definition graph will contain an edge *from* a definition of 'y' *to* a definition of 'x'. (The direction of the edge is that of the data flow.)

Of course, an event procedure may define a variable more than once, so it is important to establish which definition or definitions of a variable are 'live' at a given point; that is, can determine the value of the variable being defined. A definition is live at a given point if control can pass to it from the place where the definition was made, without the definition having being destroyed, e.g., by an intervening assignment. For example, in the sequence:

```
if z = 0 then
   y := 1;
else
   y := 2;
   y := 3;
end if;
x := y;
```

'y' is defined in three places, but control flows from only the 1st and 3rd definitions to reach the definition of 'x'. Therefore, both these definitions of 'y' should be linked to the definition of 'x' in the SDG. On the other hand, the 2nd definition of 'y' cannot reach the definition of 'x', because it is destroyed by the 3rd definition of 'y'.

It proves inconvenient to use a *set* of definitions for each variable; it is easier to give each variable at most one live definition. This is done by merging definitions wherever control flows merge. In the example just given, the live definitions of 'y' from each branch of the **if** statement are merged at its end by creating a new definition of 'y' that 'uses' them. Thus only one definition of 'y' emerges from the **if** statement to be used by 'x'. This is called the 'single definition' method of use-definition analysis.

To further simplify things, every variable may always be made to have a definition wherever it is in scope. It is assumed to be initially defined by its declaration, even if its initial definition is immediately replaced by an assignment.

How many definitions of a variable should there be? In the conventional use-definition analysis used in optimising compilers [Aho & Ullman 1972b, Kennedy 1981, Veen 1986] it is assumed that definitions correspond to points in the text of a program, typically, its assignment statements. This is obviously a simplification for programs that contain a loop, because the same variable may be freshly defined by the same assignment on each iteration. As it happens, this simplification does not affect the usual purposes to which use-definition analysis is put, but it is inadequate to deal with the question of compatibility. Clearly, we cannot construct a graph with a new vertex for every iteration of a loop, because, among other things, we cannot always predict how many iterations there will be. To understand the correct solution to this problem, we must first understand how the SDG will be used, so this part of the discussion is deferred. For the present, we ignore loops and assume that a statement creates at most one definition. We call these 'lexical' definitions, because they correspond to occurrences of the variable in the program text.

Because a variable may have several definitions, to denote one uniquely its name can be decorated with a subscript or superscript. Here, we use either 'A_2' or 'A''' to denote the second definition of 'A' within the text of the specification. When an array element is defined, it is important to know not only the name of the subscript, but also its definition, so that a definition of 'A(i)' might be denoted '$A_3(i_2)$' or '$A'''(i'')$', indicating that it corresponds to the 3rd definition of 'A(i)', indexed by the value assigned in the 2nd definition of 'i'.

8.2 The Treatment of Indexed Variables

Although loops are not discussed until a later section, it is necessary to lay some groundwork. In particular, we need to discuss the treatment of indexed variables.

For the analysis of independence, it is necessary to check if definitions are compatible. To be considered compatible, it isn't enough for two array elements to have the same index variables; they must have the same index *definitions*. Of course, two different definitions could happen to assign the same value, but the *Designer* is not smart enough to prove whether this is the case. (Only a perverse specifier would deliberately choose to assign the same value to two different index variables, but it is always possible for different index variables to have the same value by chance. Further, since a loop variable takes on every index value, it is inevitable that at some point it will replicate the value of every other index variable with the same domain.)

Indexed variables therefore have to be treated with care. One possibility, which is adequate for many applications of use-definition analysis, is to treat a definition such as 'A(i)' as defining the array 'A' as a whole. Certainly an assignment to 'A(i)' changes the value of 'A'. However, this is not adequate for checking compatibility. Instead, we regard 'A(i)' as a local variable — i.e., as if it were held in a buffer or register. If 'A(i)' is inspected and has no live definition, one is constructed from the existing definitions of 'A' and 'i'.

Even so, it is not easy to decide which edges should be part of the SDG. Consider the assignment:

```
A(i) := A(j);
```

Since 'i' and 'j' cannot be assumed to be equal, this dependence is considered to represent an incompatibility, preventing independent access. The SDG should contain an edge from the definition of 'A(j) to that of 'A(i)'. Because the indices 'i' and 'j' have different definitions, the edge causes a conflict.

Likewise, the sequence:

```
t := A(i);
i := i + 1;
A(i) := t;
```

should also cause an incompatibility. Although 'A' is indexed by 'i' in both assignments, the two occurrences of 'i' have different definitions.

However, the following case is also important:

```
A(j) := x;
y := A(i);
```

The order of the two assignments matters; there is a *possible* dependence of 'y' on 'x'. Although 'i' and 'j' probably have different values, they are not provably different. In the case that 'i' and 'j' are equal, 'y' depends on 'x'. , The process that updates 'y' cannot precede the process that inspects 'x'. In this case, even though two different index variables are involved, there is no incompatibility between 'x' and 'y', because the dependence only exists if 'i' and 'j' happen to be equal.

On the face of it, to show that the dependence exists requires the definition of 'A(i)' to use that of 'A(j)'. After all, if 'i' and 'j' are equal, the definition of 'A(i)' *is* the definition of 'A(j)'. But this case would be indistinguishable from the edge due the assignment 'A(i) := A(j)'.

This presents a dilemma. If a edge linking array elements with different indices is considered to cause incompatibility, then the SDG should not include one from 'A(j)' to 'A(i)'. On the other hand, if no edge is present, then the potential dependence of 'y' on 'x' won't be detected.

The resolution of this dilemma is to distinguish two kinds of edges. Basically, there are those representing dependences that always exist, and those representing dependences that only arise when two index variables happen to be equal. We refer to these edges as 'hard' and 'soft' respectively. Independence is determined by the hard edges, whereas separability is determined by the union of both the hard and soft edges.

A definition of 'A(j)' should not destroy the definition of 'A(i)', even though 'i' and 'j' might happen to have the same value. To see why, consider the following event specification:

```
procedure Transfer (Payer, Payee : account; Amount : money) is
begin
   Balance (Payer) := Balance (Payer) – Amount;
   Balance (Payee) := Balance (Payee) + Amount;
end Transfer;
```

When this event is invoked, 'Payer' and 'Payee' are probably different, but not provably different. If they are different, a sum of money is transferred from the 'Payer' account to the 'Payee' account. Since the operations on 'Payer' and 'Payee' are independent, they are capable of being done in parallel, as explained in Chapter 3. Briefly, if 'Payer' and 'Payee' differ, a coordinating process will typically send messages to two different processors to carry out the assignments. On the other hand, if 'Payer' and 'Payee' are the same, both messages will arrive at the same processor. But provided messages reaching a given processor are acted on in the same order that they were sent, the desired semantics will always be preserved. (In this case, executing the event procedure achieves no overall effect.)

If the analysis were to regard the assignment to 'Balance(Payer)' as affecting the whole 'Balance' array, then, since the value of 'Balance(Payee)' also depends on the 'Balance' array, the analyser would deduce that 'Balance(Payee)' depended on 'Balance(Payer)'. Now, although this is true in the case that 'Payer' and 'Payee' are equal, it is not true when they are unequal. Acknowledging a dependence would preclude the analyser finding a design in which the payer and payee could be updated independently. Therefore, it must regard the assignment to 'Balance(Payer)' as affecting only the local 'Balance(Payer)' variable, not the whole 'Balance' array. This assumption is correct when 'Payer' and 'Payee' are different, and harmless when they are the same. Therefore, a correct analysis must avoid constructing a 'hard' path from 'Balance(Payer)' to 'Balance(Payee)' in the SDG.

Even though it considers each array element separately, the analysis must not ignore the effect of the assignment to 'Balance(Payer)' on the array as a whole, because at the end of the event procedure 'Balance(Payer)' becomes undefined, but the array has certainly been updated. The value assigned to 'Balance(Payer)' by one 'Transfer' event can be the same value inspected by a later 'Transfer' event, either in the guise of 'Balance(Payer)' or of 'Balance(Payee)'. Therefore, taking the event as a whole, the use-definition graph should include edges indicating that the final definition of the 'Balance' array uses 'Balance(Payer)' and 'Balance(Payee)' — or more strictly, the definitions of them. Likewise 'Balance(Payer)' and 'Balance(Payee)' must both use the initial definition of the 'Balance' array. However, there must be no edge linking 'Balance(Payer)' and 'Balance(Payee)'. This is done by creating a new definition of the 'Balance' array at the point that 'Balance(Payer)' ceases to be live (e.g., at the end of the event procedure), dependent on the definition of 'Balance(Payer)' and the current definition of 'Balance'. The same treatment is given to 'Balance(Payee)'. The resulting use-definition graph is shown in Figure 8.2.1. This treatment is needed whenever the current definition of an index variable is dropped, i.e., when an assignment is made to it, or it passes out of scope.

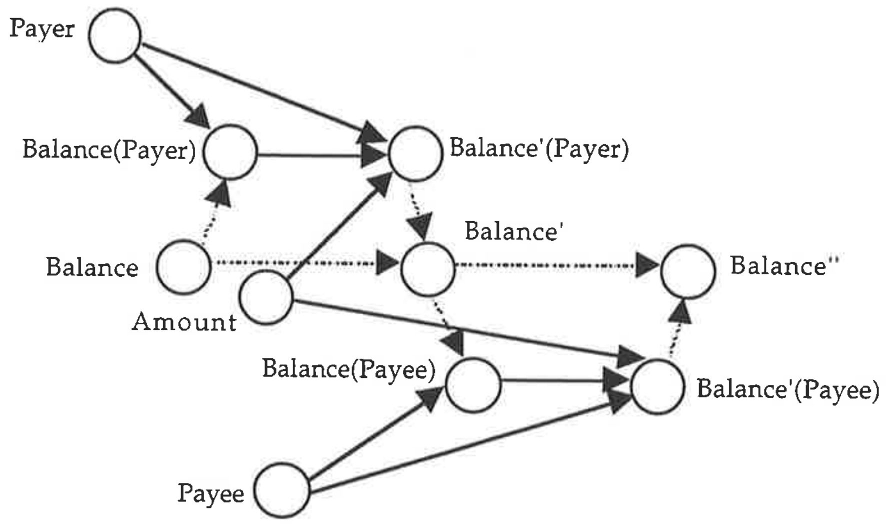

FIGURE 8.2.1: USES AND DEFINITIONS OF VARIABLES

In Figure 8.2.1, solid lines indicate 'hard' dependences, and dashed lines indicate 'soft' dependences. Basically, whole arrays only participate in soft dependences, but their elements participate in hard ones. 'Balance(Payer)' denotes the initial definition of the 'Balance(Payer)' element before its assignment, and 'Balance'(Payer)' represents its definition after the assignment. 'Balance' represents the initial definition of the 'Balance' array before the event, and 'Balance'' represents its definition after 'Payer' ceases to be live. Similarly, 'Balance(Payer)' and 'Balance'(Payer)' denotes the initial and final definitions of the 'Balance(Payer)' element, and 'Balance''' represents the final definition of the 'Balance' array after 'payee' ceases to be live. Although there is a path that includes the dashed soft edges connecting a definition of 'Balance(Payer)' with one of 'Balance(Payee)', there is no path that follows only the solid hard edges. Thus, although the graph shows the dependence of 'Balance(Payee)' on 'Balance(Payer)', it has no incompatible edges.

Standard constructions are used to model the situations when an array element is assigned a value, when it appears in an expression, and when it is dropped. Figure 8.2.2 shows that when a reference is made to an array element that is not already defined, a definition of it is created, using the current definition of its parent array and its index. Figure 8.2.3 shows that when an array element is defined by an assignment, a new definition of it is created that uses its index, and any terms that appear in the expression assigned to it — in this case, 'Balance(Payer)' and 'Amount'. No reference is made to the parent array, as it is not necessary to inspect it in order to assign the value. Finally, Figure 8.2.4 shows that when an array element is dropped, for whatever reason, a new definition is made of its parent array, using the element being dropped and the existing definition of the array. This construction may be taken to model the writing of the buffer containing the element back to its parent array.

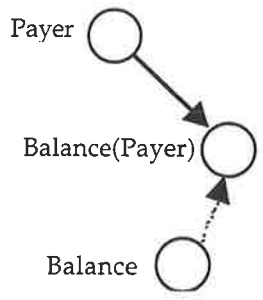

FIGURE 8.2.2: REFERENCE TO AN ARRAY ELEMENT

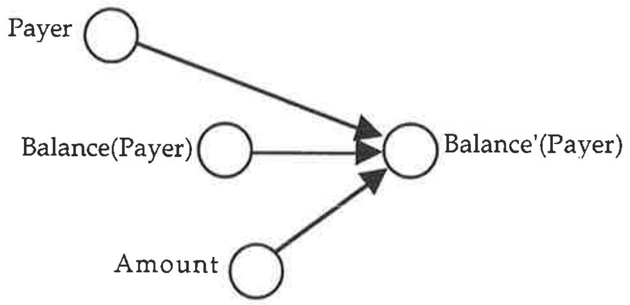

FIGURE 8.2.3: DEFINITION OF AN ARRAY ELEMENT

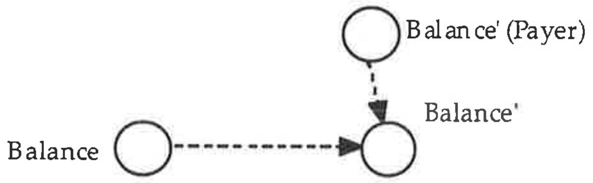

FIGURE 8.2.4: DROPPING AN ARRAY ELEMENT

At first sight, these constructions do not enforce the 5th rule of Section 4.5, which makes the array 'Balance' depend on 'Payer'. However, since 'Balance(Payer)' depends on 'Payer', and 'Balance' depends on 'Balance(Payer)' when it is dropped — which it must be sooner or later — 'Balance' depends indirectly on 'Payer'. Indeed, the constructions of Figure 8.2.2–Figure 8.2.4 seem more intuitive than the rather *ad hoc* rule of Section 4.5.

The constructions do not always result in the best possible use-definition graph. Consider a case where a 'dead-end definition' is involved, as in the following sequence:

```
A(i) := x;
y := A(i);
A(i) := z;
```

where 'A' is a state variable. The first definition of 'A(i)' does not reach the end of the sequence, and the first two occurrences of 'A(i)' could be replaced by occurrences of a local variable, so 'A' should not appear to depend on 'x'. The use-definition graph built up by the preceding constructions is given by Figure 8.2.5, from which it is seen that they cause a soft dependence of 'A' on 'x'. However, the SDG is consistent with the rewriting rules. As will be seen, without the dependence, the two definitions of 'A(i)' could finish up in different processes, and if 'A' is a state variable, there is no rewriting rule that could deal with this.

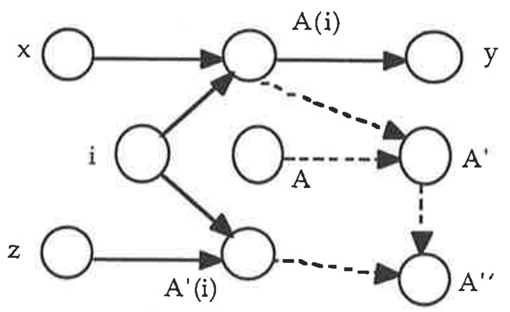

FIGURE 8.2.5: A DEAD-END DEFINITION

8.3 State Variables and Local Variables

Because the final state of a state variable generated by one execution of an event procedure becomes the initial state for its next execution, for some purposes it is necessary to link the final definitions of all state variables back to their initial definitions. This forces all definitions of a state variable to fall within the same strongly-connected component of the SDG, and so ensures that all the definitions will be made by a single process. In Figure 8.2.1, linking them will create a dependence of 'Balance' on 'Balance''', binding all the definitions of 'Balance (Payer)' and 'Balance (Payee)' into a single '{Balance}' process. In Figure 8.2.5, it will create a dependence of 'A' on 'A''', which will not bind either of the definitions of 'A(i)' into the process that updates 'A'. This is correct, the definition 'A(i)'' could occur in an earlier process that accesses 'z', and the definition 'A(i)' could occur in another unrelated process. These definitions could be made to local variables, and passed as parameters to the process that updates 'A'.

The astute reader will immediately notice that linking the final definition of 'Balance' back to its initial definition in Figure 8.2.1 would create a sneak path, from 'Balance (Payer)' through 'Balance' and 'Balance''' to 'Balance (Payee)'' — which is precisely what the analyser must avoid if it is to establish the independence of 'Balance (Payer)' and 'Balance (Payee)'. The path reflects the effect of one 'Transfer' event on another and is essential to finding the correct set of separable processes. The same value can index the 'Payer' in one transfer and the 'Payee' in the next. However, the path includes 'soft' edges (dashed lines), which will be ignored when testing to see if 'Balance' can be accessed independently.

In contrast with state variables, the final definitions of local and parameter variables are dropped at the end of an event, without linking them back to their initial definitions. This is because each execution of an event procedure uses new instances of local variables. As a result, the different definitions of local variables will not form a cycle, and may therefore be assigned to separate processes. In the case of an array declared as a local variable, because the array becomes undefined at the end of the event procedure, the final definitions of its elements must be dropped. They are not linked to the initial definition of the array, and nothing forces the accesses to the local array to be assigned to a single process. This is perfectly acceptable; a local array can be passed between processes by reference. (In practice, the array could be stored as a temporary file, and the name of the file could be passed between processes.) This differs from the treatment of state variables, for which passing a reference is invalid — because of the lags between processes, an incorrect (future) state of the variable might be observed by the downstream process. But since a local variable is private to a single instance of an event, its observed state is always exactly the one required.

It may not always be strictly correct to link the initial and final definitions of state variables. There are often restrictions on the order of events, so that a final definition created by one event may not really be the initial definition used by another, or even by itself.

The allowed sequences of events may usually be described by a regular grammar or finite-state automaton. For example, it might only be possible to close an account that was already open, it might be incorrect to open an account or close it twice in succession, and so on. A 'state transition diagram' is a graph whose vertices are states and whose transitions are events. If we consider the state transition diagram of such an account, shown in Figure 8.3.1, it has two distinct states: 'inactive' and 'active'.

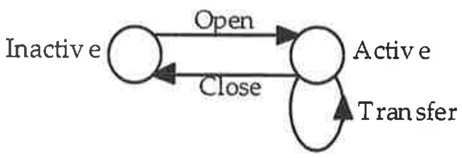

FIGURE 8.3.1: STATE TRANSITION DIAGRAM OF AN ACCOUNT

In principle, the definitions of the account attributes could be partitioned into two sets, each associated with one of the two states. Each set would contain the final definitions following the event edges that enter the states, and the initial definitions of the edges that leave them. The definitions belonging to the 'inactive' state would be the final definitions of 'Close' events and the initial definitions of 'Open' events. The definitions belonging to the 'active' state would be the final definitions of 'Open' events, and the initial definitions of 'Close' events. They would be also both the initial and final definitions of 'Transfer' events. There is then no *a priori* reason why the definitions in the 'active' and 'inactive' sets should be allocated to the same process, even two definitions of the same variable. Without changing the behaviour of the specification, a specifier could give the same attribute different names according to whether it was in the active or inactive state.

Distinguishing the states would not be helpful in this example. Consider the definitions of 'Balance' in the two states. Its definition in the 'active' state depends on its definition before the 'Open' event, which is its definition in the 'inactive' state. This definition in turn depends on its definition before the 'Close' event, which is its definition in the 'active' state. Thus the two definitions form part of the same strongly-connected component, therefore they must be accessed by the same process, and distinguishing them would achieve nothing.

The only situation where a restriction on the sequence of events is likely to offer any additional freedom to the designer is when there is an acyclic relationship between states. This might happen in following the progress of a student through a university course. The student is at first an 'applicant', after admission the student becomes an 'undergraduate', after graduation the student becomes a 'graduate'. There are no real-world events corresponding to 'de-admitting' or 'ungraduating', so the state transition diagram is acyclic. The attributes of students in these three states could be given distinct names. Indeed, it is possible that information about students in these different states might be held in different tables, or even in different databases. Despite this, imaginary events such as 'de-admitting' or 'ungraduating' would probably need to be part of the system specification, to allow errors to be corrected after a student had been 'admitted' or 'graduated' by mistake. If these were handled in the same mode that handled admitting and graduating, then separating the states would no longer be practical.

We do not address the issue of event sequence in this thesis, for three reasons: first, the specification language does not spell out the possible event sequences; second, separating distinct definitions of a variable is rarely helpful in practice; and third, when it is, the specifier can give different names to the separable definitions of variables.

8.4 An Analysis Algorithm

The analysis of an event procedure as a whole begins with the set of initial definitions of state variables. The initial definitions arising from parameter declarations and local variable declarations are added to this set, then the body of the procedure is analysed. During the analysis of the procedure body, definitions of loop variables and of array elements may be created and dropped. At the end of the event procedure, the definitions of all local variables and parameters are dropped, including any array element definitions of state variables, which are then linked to definitions of the arrays themselves.

The body of an event procedure is represented by an abstract syntax tree. Each node of the tree represents a statement. Where one statement contains other statements, the enclosed statements are part of the subtree rooted at the node that represents the enclosing statement.

An event procedure can be analysed recursively, applying local rules that depend on the type of statement represented by the root of the current subtree. Apart from the subtree itself, the recursive algorithm needs two further arguments: the set of live definitions reaching the statement, and the 'conditional context', explained in Section 8.8. As results, it returns the set of live definitions that leave the statement, and two subgraphs representing the dependences created within the statement: one representing the 'hard' edges, the other representing the 'soft' edges. The live definitions that enter a **null** statement emerge from it unchanged, but most other statements drop some definitions, and create others.

The subgraph returned by analysing the body of an event procedure represents the SDG for that event. However, since the analysis of each event begins with same set of initial definitions of state variables, the subgraphs for each event automatically become linked into a single SDG representing the complete specification.

8.5 The Analysis of Compound Statements

The body of an event procedure is a compound statement, comprising a list of simpler statements. Such a list is analysed by examining each statement in turn, applying the rules particular to each type of statement.

The analysis of the statement '**begin** S1; S2; **end**;' may be symbolised as in Figure 8.5.1. In the figure, each circle represents a set of definitions, the rectangles represent mappings from one set to another that are due to the statements concerned. The live definitions that leave the first statement in the sequence become the initial definitions for the second statement, so that the final definitions leaving the compound statement are simply those that leave the last statement in the sequence. The SDG that results from a sequence is simply the superposition of the graphs for each statement in the sequence. (The example is easily generalised to sequences of arbitrary length.)

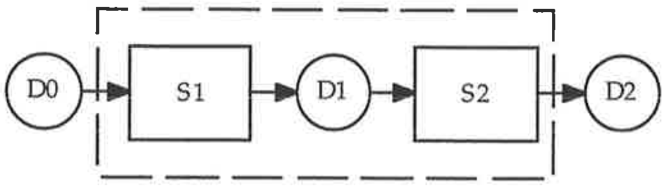

FIGURE 8.5.1: CONSTRUCTION FOR A STATEMENT SEQUENCE

8.6 The Analysis of Expressions

An expression is analysed by merging the subgraphs for each term. If a term already has a live definition, the analyser returns a subgraph consisting of a single vertex: the definition concerned. Except for indexed variables, all variables should always have one live definition. However, the first reference to an indexed variable, 'A(i)' say, should find 'A' and 'i' have live definitions, but that 'A(i)' doesn't. The analyser then creates a subgraph with a vertex for a new definition of 'A(i)' that uses the live definitions of 'A' and 'i'. (An example of this construction was used to define 'Balance(Payer)' in Figure 8.2.2 on Page 182.)

The analysis of expressions is simplified by a restriction on the specification language that requires subscripts to be simple variables rather than general expressions. It allows the analysis of terms to be completed in two passes: one to find the live definitions of simple variables, the second to find the live definitions of indexed variables. Without the restriction, the analysis of expressions would need to be recursive.

8.7 The Analysis of Assignments

The subtree representing an assignment statement has two children: one representing the assigned variable, the other representing the assigned expression. The expression is analysed (as just explained) *before* the definition of the assigned variable is dropped. Then the assigned variable is given a new definition, using the live definitions of all the terms in the assigned expression and in the 'conditional context', i.e., all the definitions used in its enclosing control expressions.

A variable is 'dropped' by deleting its current definition from the set of live definitions. An important case arises in the case of an array element. Although would be possible to keep track of several elements of the same array, in the event that their indices happened to have the same value any other element could be an alias of the element being defined. That is to say, a definition of 'A(i)' could be a definition of 'A(j)' in the case that 'i' and 'j' happen to be equal. The analyser deals with this by dropping the definition of the alias ('A(j)'), treating it as an update to the whole array, as in Figure 8.2.4 (Page 182). It then constructs a new reference to the defined variable ('A(i)'), as in Figure 8.2.2. This deals correctly with the case that the indices are equal. (It thus creates a soft dependence of 'A(i)' on 'A(j)'.) Because any existing alias is dropped when an array element is defined, there can never be more than one live element definition for any given array.

8.8 The Analysis of 'If' Statements

The treatment of **if** statements is pivotal to the analysis as a whole.

The root of a subtree corresponding to an **if** statement has three children: the control expression, the statement that is executed if the expression is true, and the statement that is executed if the expression is false. The two statements are referred to as its 'true branch' and 'false branch' respectively. Assuming that the dependence analyser is capable of processing the true and false branches recursively, it is only necessary to discuss how to combine the results from the two branches correctly.

A problem is to avoid creating false dependences. Consider the statement:

```
if U(i) then
   if V(i) then
      X(i) := W(i);
   end if;
end if;
```

where 'X(i)' depends on 'X', 'i', 'U(i)', 'V(i)' and 'W(i)'. However, 'W(i)' depends only on 'W' and 'i', and not on 'U(i)' or 'V(i)'; nor does 'V(i)' depend on 'U(i)'. The analyser should not constrain the order of accesses to 'U', 'V' and 'W' unnecessarily. The references to 'U(i)', 'V(i)' and 'W(i)' could be safely promoted outside the context of the **if** statements, as follows:

```
Ui := U(i);
Vi := V(i);
Wi := W(i);
if Ui then
   if Vi then
      X(i) := Wi;
   end if;
end if;
```

However, the reference to 'X(i)' cannot be promoted. If the control conditions fail, the final definition of X(i) is its initial definition. The following would be wrong, because there would then be no way for the initial definition of 'X(i)' to survive:

```
Ui := U(i);
Vi := V(i);
Wi := W(i);
if Ui then
   if Vi then
      Xi := Wi;
   end if;
end if;
X(i) := Xi;
```

Therefore, the analyser should make the definition created by an assignment, e.g., 'X(i)', depend on the terms in any enclosing control expressions, but it should not make the definitions of elements inspected within expressions, e.g., 'W(i)', depend on them.

The solution is as follows: During the analysis of an **if** statement, the live definitions of the terms of the control expressions are added to a 'conditional context'. When **if** statements are nested, the conditional context is extended at each level of nesting. A definition created by any assignment within the **if** statement uses all the definitions in its conditional context. For example, if a true or false branch contains an assignment statement, the analyser makes the new definition of the assigned variable depend on all the definitions in the conditional context. On the other hand, the conditional context is ignored within the analysis of expressions.

The mechanism for handling the conditional context is simple. A dummy variable is associated with the control expression, and this variable is assigned the value of the expression. The definition of this variable becomes the new conditional context. Since its evaluation takes place under the control of the existing conditional context, the new context automatically depends on the existing one. As a rule, representing the conditional context by a single variable will generate less edges in the SDG than representing it by the set of variables in the control expression would do. If there are m terms in the control expression and n assignments dependent on it, using the dummy variable will require $m+n$ edges; without the dummy variable there would need to be $m \times n$ edges.

As has already been outlined, rather than have two definitions of the same variable emerge from an **if** statement, they are merged into a single definition. Each branch is capable of creating new definitions for variables. If a branch does not define a variable, the same definition that entered the branch will also leave it. Since the set of definitions entering each branch is the same, the two definitions of a variable that leave the branches will be the same only if neither branch created a new one. But if the definitions are different, the analyser generates a subgraph containing a new definition that depends on both those emerging from the branches. Since definitions made within the branches depend on the conditional context, which includes the **if** expression itself, the new definition indirectly depends on the **if** expression.

The construction for the statement '**if** B **then** S1; **else** S2; **end if**;' is symbolised in Figure 8.8.1. The set of definitions entering the **if** statement is augmented by assigning the control expression to the dummy variable representing the conditional context. The augmented set of definitions ('D1') forms the initial set of definitions for both the true and false branches. 'D2' and 'D3' are the sets of definitions emerging from the two branches. These two sets are then merged into one to produce 'D4'.

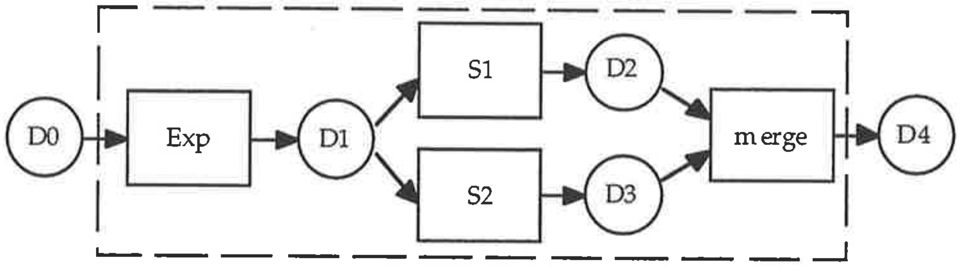

FIGURE 8.8.1: CONSTRUCTION FOR AN 'IF' STATEMENT.

It is easy to merge the definitions of simple variables. If the variable is assigned a value in either branch, its two definitions will be different. A new, third definition is constructed which then uses the other two, so that only one definition emerges from the **if** statement. Since any assignment within the **if** statement will make the assigned variable depend on the conditional context, the new definition also indirectly depends on the context. However, if the definitions emerging from the true and false branches are the same, then the variable cannot have been assigned a value, and no construction is needed.

Merging definitions is more complicated when a conditional assignment is made to an indexed variable, 'A(i)' say. If the index variable ('i') has the same definition emerging from each branch, the definitions of the indexed variable can be merged as above. A further case arises where no reference has been made to 'A(i)' before the **if** statement, one branch of the **if** statement refers to 'A(i)', but the other doesn't, and so no definition of 'A(i)' emerges from it. In that situation, the definition emerging from the second branch should actually be that which would have held on entry to the **if** statement. Logically, the analyser should construct the missing definition, and merge it with the live definition from the active branch. However, it is simpler to let the existing live definition emerge from the **if** statement, because creating the new one would add no fresh dependences to the SDG. (It would merely cause 'A(i)' to depend on 'A' and 'i', which it does already.)

However, a more careful treatment is needed when the index has a different definition emerging from each branch, as in the following example:

```
if E then
   i := j;
   t := A(i);
else
   i := k;
   t := A(i);
end if;
B(i) := t;
```

Here, there are three definitions of 'i', two created by assignments, and the merged definition following the **if** statement, which is used in the assignment to 'B(i)'. The two definitions of 'A(i)' potentially refer to different elements of 'A'. It is therefore illogical to merge them into a single definition. It would be feasible to create a new definition of 'A(i)' using the merged definition of 'i', and make it depend on the two existing definitions created by the assignments. However, this definition would be incompatible with both existing definitions, and would suggest that independent access to 'A' was not possible, when it actually is. So instead, it is better to link both definitions of 'A(i)' by 'soft' edges to a new definition of the array 'A' as whole, then drop them. This does not create the appearance of an incompatibility, because it does not create a path between the two definitions of 'A(i)'.

However, the 'single definition' method of analysis described here does not solve all compatibility problems. For example, the definition of 'B(i)' above uses the merged definition of 'i', and so appears to be incompatible with both definitions of 'A(i)' because all three have different index definitions. In reality, it is compatible with both of them: whichever path the execution takes, the value of 'i' used in 'B(i)' is the same as that used in 'A(i)'.

Two other methods of analysis can be considered: the 'set of live definitions' method, and the 'case analysis' method.

In the 'set of live definitions' method, which is well known and widely used [Aho *et al.* 1986], a variable is allowed to have multiple definitions. Thus, after an **if** statement, the sets of definitions from its two branches are simply merged together, forming their union. Immediately after the **if** statement in the above example, there would be two definitions of 't', two definitions of 'i', and two definitions of 'A(i)'. Unfortunately, it is not possible to recognise which definitions are associated with each another. The reference to 'B(i)' must therefore use both definitions of 'i', creating two alternative definitions of 'B(i)'. Each definition of 'B(i)' must also use both definitions of 't', indirectly using both definitions of 'A(i)'. Two combinations of 'B(i)' with 't' are compatible, but the other two create the appearance of incompatibility. The 'set of live definitions' method therefore has no advantage over the 'single definition' method, and tends to generate a more complex SDG.

On the other hand, the 'case analysis' method is capable of analysing the above example correctly. This method considers the two possible executions of the **if** statement separately. If the control condition is true, only one definition of 'i' and one definition of 'A(i)' emerge, and the definition of 'B(i)' is found to be compatible with that of 'A(i)'. The same conclusion is reached if the control condition is false. The two cases may be considered by back-tracking through the possible execution paths. In effect, two definitions of 'B(i)' may be constructed, one for each execution path. Essentially, the 'single definition' and 'set of definitions' methods fail because they construct only one case for the definitions 'B(i)', which of course cannot be compatible with both definitions of 'A(i)'. However, the 'case analysis' method is not infallible, as the following example shows:

```
if E then
   i := j;
   t := A(i);
else
   i := k;
   u := A(i);
end if;
if E then
   B(i) := t;
else
   B(i) := u;
end if;
```

Unless the analysis recognises the equality of expressions (the two occurrences of 'E'), it will consider that there are four execution paths rather than just two. The two cases that cannot occur in practice create apparent incompatibilities. A second problem with the 'case analysis' method is that, as in the above example, the number of execution cases can increase exponentially with the number of **if** statements.

Given that all three methods will sometimes find false incompatibilities, the 'single definition' method was chosen because it has the least computational complexity. Given suitable supporting data structures, its worst-case execution time is proportional to the product of the length of the program text and the number of live variables. Although the 'case analysis' method will succeed in some cases when the 'single definition' method fails, it is doubtful whether such cases arise often enough in practice to justify its use.

One final point. To satisfy the needs of the *Programmer* tool it is necessary to associate the **if** statement with a process. This is done by treating the evaluation of the control expression as an assignment to a dummy variable. Dependence analysis and optimisation result in allocating this variable to a process. This process is then the correct one in which to evaluate the control condition, and therefore the correct place for the **if** statement.

8.9 The Analysis of Procedure Calls

The analyser does not support internal procedures. The effects of internal procedures can be specified by writing them in-line, or replacing them by one or more function calls. The only procedure calls allowed are those on events in external packages.

The subtree representing a procedure call has two children: the identifier of the called procedure, and a list of all the terms in the expressions that are the parameters of the call. A **call** statement is treated as if it were an assignment in which the parameters are assigned to the called procedure. However, if a live definition of the called procedure already exists, it is *not* dropped, but is used by the new definition, as if the existing definition were another parameter.

It is necessary because any definitions that are not live at the exit of an event procedure are ignored, as they can have no external effect. If a definition created by a procedure call did not use the existing definition of the called procedure, it would be treated like a dead-end assignment, and only the final call would have any effect. This approach can also be justified by considering the called procedure to be associated with a queue to which successive calls are appended.

Because calls are treated like assignments, they also receive exactly the same treatment within **if** statements.

8.10 The Analysis of Return Statements

The subtree representing a **return** statement has one child: the returned expression. There are two contexts in which a **return** statement can appear, within a function, or within a procedure. If the **return** statement appears in a function, it is not analysed at all, because the analysis of the whole function is skipped. A function call such as 'min(x,y)' is treated in the same way as the expression 'x+y'; the actual definition of the function is irrelevant to dependence analysis. A **return** statement within a procedure can't return an expression. However, since a **return** statement causes the procedure to exit, it is treated in the same way as the end of the procedure itself: the definitions of local variables are dropped and the definitions of array elements are linked to final definitions of their parent arrays.

8.11 The Analysis of Loops

If a reference is made to an array element such as 'A(i)' and a reference is made to a different element such as 'A(j)', we assume that 'i' and 'j' can be unequal. If 'A(i)' depends on 'A(j)' or 'A(j)' depends on 'A(i)', either directly or indirectly, we say the references to 'A' are incompatible. Similarly, if 'A(i)' depends on 'A(i)' but 'i' has different definitions in each case, we assume the definitions can be unequal, and again decide that the references to 'A' are incompatible. What happens if the references occur inside a loop that modifies 'i'? Although 'i' is known to have different values on each iteration of the loop, it only has one definition, because the definition of 'i' corresponds to one of its lexical occurrences in the text of the loop. A definition cannot be unequal to itself.

To analyse such loops properly, it is necessary to distinguish 'dynamic definitions' from 'lexical definitions'. That is, although 'i' may have only one *lexical* definition in the text of the specification, it is given a new *dynamic* definition for each iteration of the loop. Each lexical definition therefore corresponds to a series of dynamic definitions. In the absence of loops, each lexical definition has at most one dynamic definition, which is why it was unnecessary to distinguish them in the preceding discussion.

However, it would be wrong to drop the notion of lexical definitions in favour of dynamic definitions, because it is lexical definitions that must be allocated to processes. It would be embarrassing for two dynamic definitions to be allocated to different processes, only to find that they were associated with the same lexical definition. This would imply that the same fragment of program text would need to be placed in more than one process, which would be inconsistent with the rewriting rules for delayed procedure call. Therefore, it is necessary to use lexical definitions to determine separability, but to use dynamic definitions to analyse independence.

8.11.1 While Loops

The most difficult case is that of a **while** loop. The analyser must try to determine what dependences exist between one iteration of the loop and the next. We may immediately distinguish two kinds of **while** loop. The first is illustrated by Example 7.3.1, reproduced here as Example 8.11.1, which sets 'Order_Qty (What)' to the least multiple of 'Economic_Order_Qty (What)' that is no fewer than 'Shortage (What)'.

```
Order_Qty (What) := 0;
while Order_Qty (What) < Shortage (What) loop
   Order_Qty (What) := Order_Qty (What) + Economic_Order_Qty (What);
end loop;
```

EXAMPLE 8.11.1: A 'COMPATIBLE' WHILE LOOP

As already discussed, the loop in Example 8.11.1 has little effect on dependence analysis. The loop involves only one value of 'What', so there can be no interaction between different array elements.

The second kind of loop is illustrated by Example 7.3.2, reproduced here as Example 8.11.2.

```
while Boss (Who) /= null loop
   Who := Boss (Who);
end loop;
```

EXAMPLE 8.11.2: AN 'INCOMPATIBLE' WHILE LOOP

The loop in Example 8.11.2 accesses multiple values of 'Who', and furthermore, each value depends on the previous one. For this dependence to be apparent in the SDG, there must clearly be more than one vertex representing a definition of 'Who', and they must be connected by a path.

If we assume that definitions are associated with places in the text of the specification, there is only one relevant definition of 'Who', which must clearly be equal to itself. However, if we assume that the number of definitions of 'Who' is equal to the number of iterations of the loop, the number is unknown, and may even be infinite. For the *Designer* program to succeed, the SDG must be finite, so some way has to be found to represent a potentially infinite graph in finite space. The graph must obviously contain at least two definitions of 'Who', otherwise — since a thing must be equal to itself — there is no way to conclude that its definitions on successive iterations are unequal.

A potential problem is that it may take more than two iterations to establish a dependence between variables, as in Example 8.11.3.

```
while A /= 0 loop
   A := B;
   B := C;
   C := D;
end loop;
```

EXAMPLE 8.11.3: AN INDIRECT DEPENDENCE

It takes 3 iterations of the loop in Example 8.11.3 to establish that 'A' depends on 'D'. The first definition of 'C' depends on the initial definition of 'D', the second definition of 'B' depends on the first definition of 'C', and the third definition of 'A' depends on the second definition of 'B', and thence of the initial definition of 'D'. In principle, such a chain of assignments may be arbitrarily long.

A possible approach is to regard a **while** statement as an sequence of **if** statements: i.e., the statement '**while** E **loop** S; **end loop**;' is treated as '**if** E **then** S **end if**; **if** E **then** S **end if**; ...'. The SDG is constructed for at least two successive **if** statements, but the construction must continue until all dependences have been discovered. The SDG for two iterations of Example 8.11.3 is shown in Figure 8.11.1.

In Figure 8.11.1, unprimed names, such as 'A', represent the initial definitions of the variables before the loop. The primed names, such as 'A'', represent the definitions created by the assignments on the first iteration of the loop. All of them depend on 'A', because 'A' is part of the loop's control expression. The doubly-primed names, such as 'A''', represent the merged definitions that result from treating the first iteration of the loop as an **if** statement. The triply-primed names, such as 'A'''' represent the definitions created on the second iteration of the loop. The quadruply-primed names represent the merged definitions resulting from treating the second iteration as an **if** statement. They all depend on 'A'''. This construction correctly models the dependence of 'A' on 'C': there is a path from the initial definition 'C' through 'B'', 'B''', and 'A'''' to 'A''''', the final definition of 'A'. However, it would take a third iteration to establish the dependence of 'A' on 'D'.

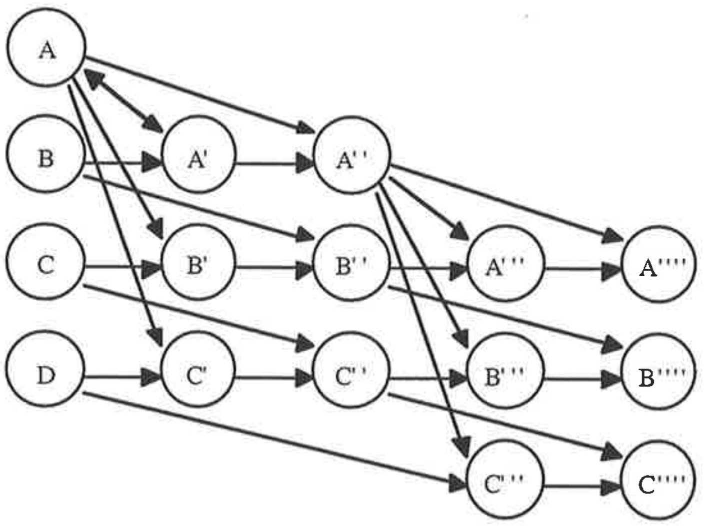

FIGURE 8.11.1: SDG FOR 2 ITERATIONS OF EXAMPLE 8.11.3

How can it be determined when enough iterations have been modelled? It seems intuitively obvious that termination should be associated with finding the closure of the SDG. However, the graph of dynamic definitions can never reach closure, because each iteration generates a new set of vertices. Instead, we turn to the graph of lexical definitions, shown in Figure 8.11.2.

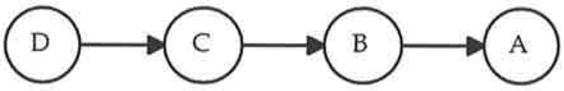

FIGURE 8.11.2: LEXICAL DEPENDENCES FOR ONE ITERATION OF EXAMPLE 8.11.3.

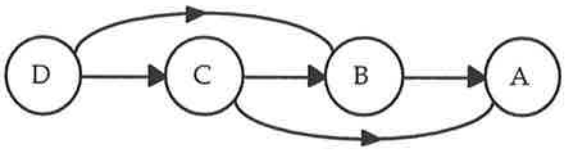

FIGURE 8.11.3: LEXICAL DEPENDENCES FOR 2 ITERATIONS OF EXAMPLE 8.11.3.

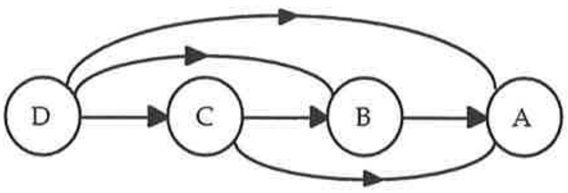

FIGURE 8.11.4: LEXICAL DEPENDENCES FOR 3 ITERATIONS OF EXAMPLE 8.11.3.

Figure 8.11.2 shows the dependences between lexical definitions that can arise in one iteration of the loop. Figure 8.11.3, shows the dependences that arise after two iterations of the loop. This graph is found by forming the product of the original graph with itself. The product contains an edge for every compound path of length 2 in the original graph. The union of the product and the original graph yield Figure 8.11.3. Similarly, Figure 8.11.4 shows the dependences arising from 3 iterations. This is found by forming the product of the original graph with Figure 8.11.3, which contains an edge for every path of length 3 in the original graph, then forming its union with Figure 8.11.3. However, if we were to repeat this process again, the resulting graph would still be that of Figure 8.11.4, because the longest path in Figure 8.11.2 has length 3. In other words, Figure 8.11.4 is the transitive closure of the lexical SDG. Since it takes 3 steps to find the closure of the lexical graph, modelling 3 iterations of the dynamic definition graph is also sufficient to model all the dependences between variables.

A more appealing idea is to model the loop by constructing a dynamic definition graph that also contains a loop. With the correct construction, only two definitions of the variables are ever needed to represent any loop correctly. The construction is shown in Figure 8.11.5. Each iteration is modelled by an **if** statement, as before. The set of definitions emerging from the second iteration of the loop body is linked back to the set of definitions entering it. This simulates as many iterations of the loop as needed. The first iteration must still be treated separately, for two reasons: at least two dynamic definitions are needed to detect incompatibility, and it prevents unwanted interactions between different loops.

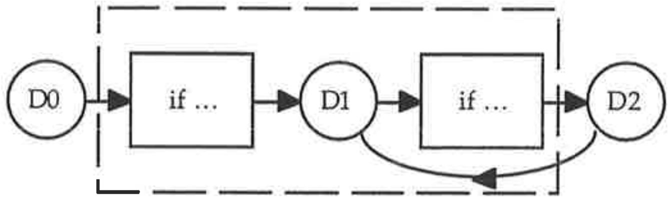

FIGURE 8.11.5: AN ALTERNATIVE CONSTRUCTION FOR A 'WHILE' LOOP.

The difficulty with this approach is to decide which definitions should be linked together. Linking identical dynamic definitions would simply link them to themselves. Linking identical lexical definitions would also be incorrect, because there may be several lexical definitions of a variable within a loop, so its final lexical definition should be linked to its first. The only remaining alternative seems to be to link the live definitions of the same variables. Whereas this is correct, we must be careful about what we mean by 'the same variable'. Consider the following loop:

```
i := 0;
while i < n loop
   i := i + 1;
   A(i) := A(i) + 1;
end loop;
```

It is certainly sensible to link the final and initial definitions of 'i', because its final definition for one iteration is its initial definition for the next. Unfortunately, applying the same rule to the definitions of 'A(i)' links two definitions of 'A(i)' having different dynamic definitions of 'i' as indices, so the accesses to 'A' seem to be incompatible. Clearly, it is wrong to regard the two definitions as matching, because they can concern different array elements. The rule adopted is therefore the following: two definitions match either if they are definitions of the same simple variable, or they are definitions of the same array element, i.e., they have identical index definitions. In this case, the definitions of 'A(i)' would not be linked, because, although they are both indexed by the same lexical definition of 'i', they are indexed by different dynamic definitions.

As an example, consider the loop of Example 8.11.2. In Figure 8.11.6 'Who' and 'Boss' are the initial definitions of 'Boss' and 'Who' before the loop. The (unprimed) definition 'Boss (Who)' is that caused by the reference to 'Boss' in the expression controlling the first execution of the loop. The (primed) definition 'Who'' is created by the first execution of the assignment statement, and depends on 'Boss (Who)' both because it is the assigned expression, and because it is in the conditional context. Since the existing definition, 'Who', must be dropped, 'Boss (Who)' becomes invalid, and must be dropped too. It is linked to a new definition of the 'Boss' array, 'Boss'', which depends also on 'Boss'. The (doubly-primed) definition 'Who''' results from conditionally merging the definitions of 'Who' and 'Who''. The definition 'Boss''' results from conditionally merging the definitions of 'Boss' and 'Boss''. In a similar way, the definitions of 'Boss''(Who'')', 'Who''', 'Who''''', 'Boss''' and 'Boss''''' are constructed to model the second iteration of the loop. (Note the edge from 'Who''''' back to 'Who'''. Because 'Boss''(Who'')' indirectly depends on 'Boss(Who)'', and 'Who''' and 'Who' are different definitions, we deduce that the accesses to 'Boss' involve different indices, and are not independent.

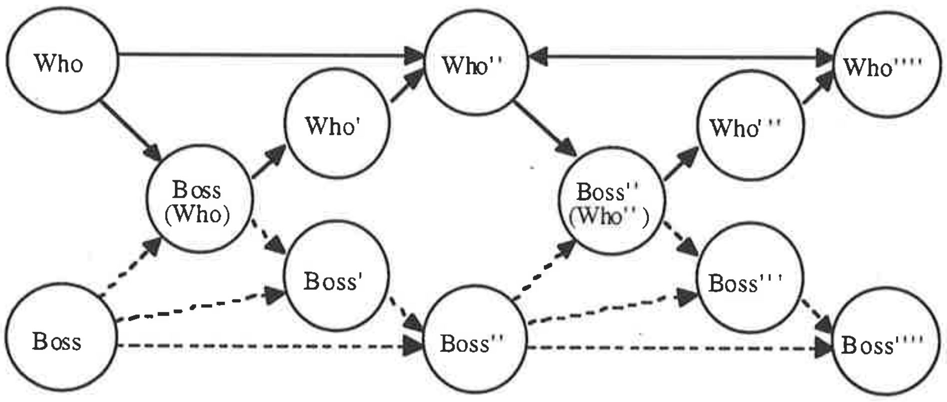

FIGURE 8.11.6: THE SDG FOR THE LOOP OF EXAMPLE 8.11.2.

On the other hand, in the case of the loop of Example 8.11.1 it is clear that, whatever its SDG may look like, there is only one definition of 'What', so all the accesses to 'Shortage', 'Economic Order Qty' and 'Order Qty' are compatible. Therefore, the final definition of 'Order_Qty(What)' should be linked to the initial one.

8.11.2 All Loops

The treatment of **all** loops depends on whether the system implementation will use sequential or parallel access. If sequential access is intended, an **all** loop can be treated in exactly the same way as a **for** loop, discussed in the next section. There are two approaches to the analysis of parallel **all** loops: the 'trusting', and the 'careful'. Using the trusting approach, since the specifier has declared the instances of the loop body to be independent, they may be treated just like a single instance.

The trusting approach is not always safe. For example, in the loop:

```
all i in index loop
    T := A(i);
    B(i) := T;
end loop;
```

EXAMPLE 8.11.4: A BADLY SPECIFIED ALL LOOP

there is a potential interaction between loop body instances via the shared variable 'T'. The effect of the loop is unpredictable, and is probably an error on the part of the specifier. A correct analysis should find that 'B(i)' can use an instance of 'A(i)' with a different definition of 'i'.

An **all** loop may more accurately be modelled by two parallel executions of its loop body, each using a different definition of its loop variable. Then every use of a shared variable (i.e., any variable not indexed by the loop variable) in one execution instance should be cross connected to *every* definition of the same variable in the other instance. This models all possible timings of the two execution paths. Figure 8.11.7 shows the SDG for the loop of Example 8.11.4. (For ease of drawing, some edges have been omitted.)

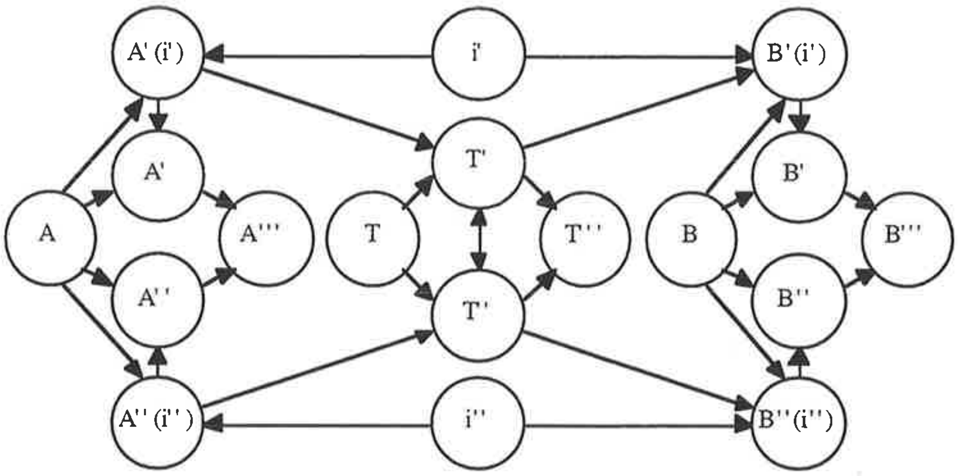

FIGURE 8.11.7: SDG FOR THE LOOP OF EXAMPLE 8.11.4.

In the SDG of Figure 8.11.7, unprimed names represent the initial definitions of the corresponding variables, singly primed names represent definitions associated with a first loop body instance, doubly primed names represent definitions associated with a second loop body instance, and triply primed names represent the final definitions obtained by merging the two loop body instances. Because 'T' is a shared variable, 'T'' and 'T''' are interconnected (but not 'A'' and 'A''' or 'B'' and 'B'''). This creates a path from 'B'(i')' to 'A''(i'')', for example, which involves different index definitions, revealing that the loop cannot be safely executed in parallel.

This treatment does not forbid shared variables, providing they are used correctly. In particular, if the shared variable is read-only, no path involving unequal indexes will occur, nor will one occur in the case of an accumulator variable.

Actually, the current implementation of the *Designer* tool does not use either of these approaches, but treats an **all** loop as if it were a **while** loop containing an assignment to the loop variable. The reason for this arises from the restriction that indices must be simple variables, a property that simplifies many aspects of the *Designer*. As was illustrated in Example 2.6.1 and Example 2.6.2, this sometimes means that index variables must be declared within a loop body. However, the *Designer* is not sophisticated enough to distinguish a correct situation, as in Example 2.6.2, from an incorrect situation, as in Example 2.6.1. Correcting the defect would be laborious, but it is not a problem in principle.

8.11.3 For Loops

Unlike an **all** loop, a **for** loop may reliably create a dependence between one indexed element and another, as in Example 8.11.5. This loop only makes sense in a sequential context, never in a parallel one. Indeed, it is unlikely to be sensible in most sequential contexts too, because there is usually no logical connection between the successive values of such indexes as customer numbers or product codes.

```
for i in index loop
   T := T + A(i);
   B(i) := T;
end loop;
```

EXAMPLE 8.11.5: A DEPENDENCE BETWEEN ITERATIONS

In a parallel implementation, a **for** loop must be treated as a **while** loop containing an assignment to the loop variable. Thus the loop variable has a different definition in each instance of the loop body. But in a sequential implementation, it may be treated as a **while** loop in which the loop variable is given a definition before the loop, rather than within the loop. The loop variable then has the same definition in each loop body instance, so that paths between definitions in successive iterations will not destroy compatibility. This is the method currently used by the *Designer*. As a result, the *Designer* can be used to derive designs for either parallel or sequential implementation by changing the way that **for** loops are specified. In most cases a **for** loop can be expressed as an **all** loop. But in those cases where there is intended to be a dependence between iterations, as in Example 8.11.5, an **all** loop cannot be used. If a sequential implementation is intended, a **for** loop should be specified, but if a parallel implementation is intended, a **while** loop must be specified instead.

9. Optimisation

9.1 The Composition Problem

Once the SDG of a system has been derived from a specification, it is simple to find the strongly-connected components that define its CPG. There are well-known algorithms that can find the strong components of a graph in a time proportional to the size of the graph. (*See* Section 4.8.) The CPG always allows a real-time equivalent decomposition of the system into component processes, but it is not always the most efficient decomposition. As illustrated in Chapter 6, it often pays to combine compatible processes.

The processes of a system are composites of its canonical processes, chosen to optimise system performance. Each composite process corresponds to a set of canonical processes, and each canonical process belongs to exactly one composite. In other words, the composite processes partition the vertices of the CPG.

Even a small set has many possible partitionings. For example, if a process graph has just 5 vertices, they may be partitioned into composites with the following sets of sizes: {1,1,1,1,1}, {1,1,1,2}, {1,1,3}, {1,2,2}, {1,4}, {2,3}, or {5}. There are 52 partitionings in all: there is 1 way to form the {1,1,1,1,1} or {5} partitionings, 10 ways to form {1,1,1,2}, {1,1,3} or {2,3} partitionings, 15 ways of forming {1,2,2} partitionings, and 5 ways of forming {1,4} partitionings. The number of ways of partitioning a set grows very rapidly with the size of the set, being a Stirling Number of the second kind. Although the examples in earlier chapters suggest that an optimal set of composite processes is easily found by common sense, it will be shown that the optimum partitioning problem is potentially complex — specifically, NP-complete.

No computer algorithm can afford to test all the possible partitionings of a large process network. A more directed search for a solution is needed. Two methods are considered here: Branch and Bound Search, and a greedy heuristic method. Both methods have their own characteristics. Branch and Bound Search guarantees to find a solution that has the least value of some cost function, in this context, some measure of execution time. Although a Branch and Bound Search often finds a optimal solution quickly, its worst-case execution time grows rapidly with the size of the problem. On the other hand, a greedy heuristic method always finds a solution quickly, but the solution may not be optimal. In practice, a greedy method can be found that performs surprisingly well for realistic process composition problems.

9.2 The Lattice of Process Graphs

Both the methods to be discussed operate by combining pairs of processes, starting from the CPG. It is important to show that an optimal solution can indeed be found by pair-wise composition, and that no more general form of composition is needed. This can be proved by showing that the set of all partitionings forms a lattice under pair-wise composition.

A partitioning generates a homomorphism from the canonical minimal process graph to a 'composite process graph' in which sets of vertices (minimal processes) are mapped to a single partition (a composite process). Each partition is represented by a vertex of the composite process graph. An edge joining two vertices of the canonical graph is mapped to an edge joining the corresponding partitions of the composite graph. Duplicate edges are merged.

Composing the vertices of an acyclic graph in this way may generate a cyclic graph. Because of the Data Flow Theorem, cyclic graphs do not correspond to valid process graphs. (Such an invalid composition was illustrated in Figure 4.7.4 of Page 93.) Since all valid process graphs are acyclic, they have unique transitive roots. Therefore, if we consider only their transitive roots, each partitioning of a CPG that leads to a valid process graph has a unique corresponding composite process graph.

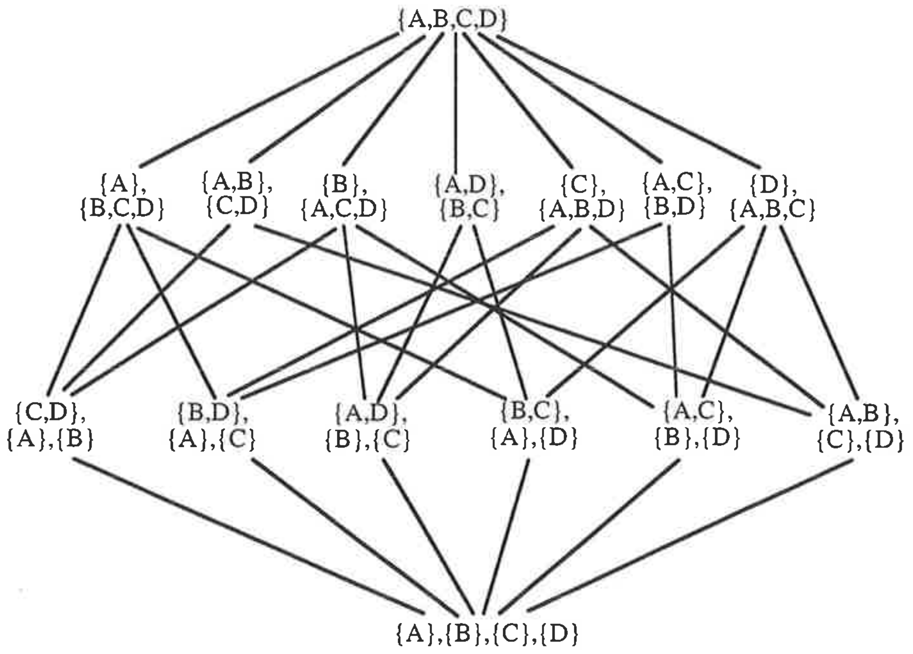

FIGURE 9.2.1: THE LATTICE OF PARTITIONS FOR 4 ELEMENTS

Figure 9.2.1 shows the lattice of partitionings of the set {A,B,C,D} generated by pair-wise compositions. The elements 'A', 'B', 'C' and 'D' represent the separable processes of some system. The partition '{A,B}' denotes the composite process generated by combining '{A}' and '{B}' to access both their associated sets of variables. The partitioning '{C,D},{A},{B}' denotes the family of process graphs that have {A}, {B} and {C,D} as component processes and preserve the partial ordering of the canonical minimal separable process graph.

The graph has four levels, which correspond to the numbers of partitions. The top element comprises the single partition '{A,B,C,D}'; the bottom element contains the four partitions '{A}', '{B}', '{C}', and '{D}'. (The order of the terms is unimportant; '{C,D},{A},{B}' and '{A},{B},{C,D}' denote the same partitioning.) An edge between two partitionings indicates that the upper partitioning can be generated from the lower one by composing two of its partitions.

The relation 'can be generated by zero or more pair-wise compositions' defines a partial ordering, because it is clearly reflexive, transitive and antisymmetric. For the partially ordered set to form a lattice, every pair of partitionings must have a least upper bound and a greatest lower bound. The least upper bound $P \vee Q$ of two partitionings P and Q is such that, for each pair of elements x and y, x and y belong to the same partition of $P \vee Q$ if and only if they belong either to the same partition of P or to the same partition of Q. Likewise, their greatest lower bound $P \wedge Q$ is such that for each pair of elements x and y, x and y belong to the same partition of $P \wedge Q$ if and only if they belong both to the same partition of P and to the same partition of Q. Clearly, '{A,B,C,D}' is a universal upper bound, as it always possible to merge partitions pair-wise until it is reached. Likewise '{A},{B},{C},{D}' is a universal lower bound, because its partitions can always be merged to generate any desired partitioning. Therefore every pair of partitionings has a least upper bound and a greatest lower bound.

The existence of the lattice justifies the use of optimisation methods based on merging partitions in pairs. The optimisation problem may be visualised by labelling the vertices of the lattice with processing costs. An optimal solution is represented by a vertex with the lowest cost. The universal lower bound vertex represents the CPG. It will not usually be the optimum partitioning, because it maximises the number of separate accesses needed to variables and the number of data flows. The universal upper bound vertex represents a solution in which a single process accesses all the state variables, i.e., it has the same form as the specification. It too will not usually be the least cost vertex, because it will typically contain sets of conflicting variables, and will not allow independent access. Along the paths from the universal lower bound vertex to an optimal partitioning the cost function usually improves monotonically. For example, if the optimum solution is '{A,B,C},{D}', then the costs of '{A,B},{C},{D}' and '{A},{B,C},{D}' typically lie somewhere between the cost of the CPG '{A},{B},{C},{D}' and the optimum. This is the basis of the heuristic method to be described shortly.

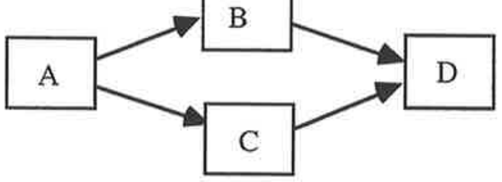

FIGURE 9.2.2: A CANONICAL MINIMAL PROCESS GRAPH

A complication in finding an optimum solution is that not all partitionings correspond to acyclic process graphs. Consider the CPG of Figure 9.2.2. For this graph, the partitioning '{A,D},{B},{C}' would not correspond to a valid process graph because the {A,D} process would have cyclic data flows with both the {B} and {C} processes. Suppose the optimal partitioning is actually '{A,B,C,D}'. This can be generated by two pair-wise compositions from the invalid '{A,D},{B},{C}' partitioning. However, it is also possible to reach it along the path '{A},{B},{C},{D}', '{A},{B,C},{D}', '{A,B,C},{D}', '{A,B,C,D}' passing only through feasible process graphs. Is it always possible to generate a valid partitioning by such a sequence of feasible compositions? Searching among feasible partitions would reduce the search space.

Fortunately, the answer is yes. The composition of two processes is always infeasible if the longest path joining them in their associated process graph has more than one edge. Since any compound path between them must include at least one other process, combining two processes linked by a compound path must create a data flow cycle. However, it is always safe to combine two processes that are linked by a single edge, or that are not linked at all, which we call 'feasible compositions'. Feasible compositions cannot create cycles.

To show that infeasible compositions are unnecessary, consider the proposition that an infeasible composition is needed to reach the optimum partitioning. Since the optimum partitioning must be valid, it can contain no data flow cycles. Therefore the vertices of any cycle that is generated by an infeasible step must eventually be merged into a single process in the optimum partitioning. Consider that the infeasible composition combines processes P_0 and P_n, and that there is some longest compound path $P_0, P_1, \ldots P_{n-1}, P_n$ linking them, where $n \geq 2$. (P_1 is not necessarily distinct from P_{n-1}.) Then $P_0, P_1, \ldots P_{n-1}, P_n$ will all belong to the same partition in the optimal partitioning. Since P_1 is the first process on a longest path from P_0 to P_n, the longest path from P_0 to P_1 is a single edge. Combining P_0 with P_1 is a feasible composition. By finding the longest path between $\{P_0,P_1\}$ and P_n in the resulting process graph and repeating the process, all the processes linking P_0 to P_n can be combined with P_0, even if there are multiple paths. Thus is never necessary to consider infeasible compositions.

For example, given the desire to compose {A} and {D} in Figure 9.2.2 infeasibly, their composition would create a cycle containing {A,D} and {B} and a cycle containing {A,D} and {C}. Therefore, all the vertices would be forced into the same separable process, {A,B,C,D}. But the sequence, '{A},{B},{C},{D}', '{A,B},{C},{D}', '{A,B,C}, {D}', '{A,B,C,D}' could construct {A,B,C,D} using only feasible compositions. There are two longest paths between {A} and {D}, one passing through {B}, the other through {C}. Two compositions are suggested: {A} with {B}, or {A} with {C}. After arbitrarily choosing the {A,B} composition, the resulting transitive root still contains a compound path from {A,B} to {D}, passing through {C}. Therefore {C} is composed with {A,B}, after which the composition of {A,B,C} with {D} becomes feasible.

The existence of a sequence of feasible compositions that can generate any feasible partitioning implies that the set of feasible process graphs also forms a lattice under feasible composition. Figure 9.2.3 shows the lattice for the CPG of Figure 9.2.2.

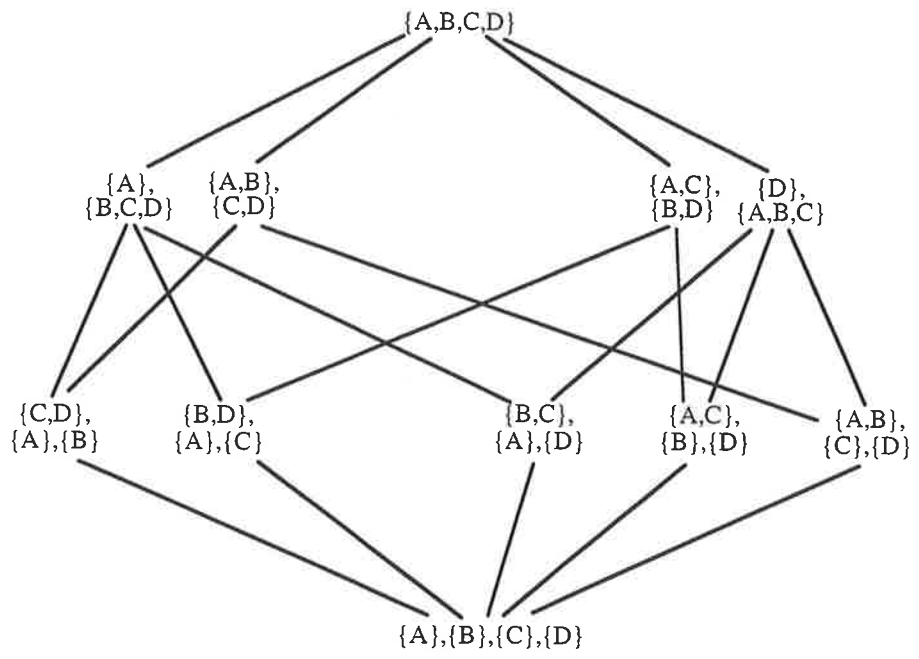

FIGURE 9.2.3: THE LATTICE OF FEASIBLE PROCESS GRAPHS FROM FIGURE 9.2.2

9.3 Two Example Problems

Two related examples will be used to illustrate optimisation. They result from slightly different versions of the same specification, which appeared in Example 6.6.1 and Figure 6.6.1 (Page 144). Their SDG's are reproduced in Figure 9.3.1 and Figure 9.3.2. The graph of Figure 9.3.1 is acyclic and leads to an efficient independent access implementation; that of Figure 9.3.2 is cyclic, and has no efficient independent access implementation.

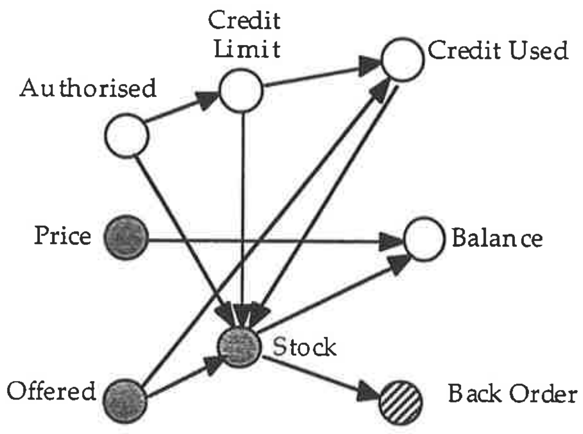

FIGURE 9.3.1: AN ACYCLIC SDG

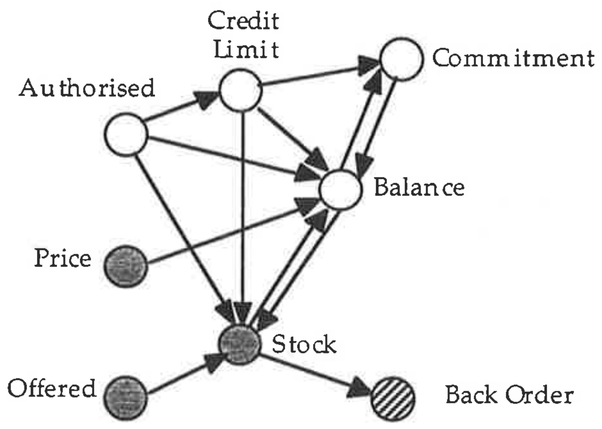

FIGURE 9.3.2: A CYCLIC SDG

The first step is to derive their corresponding CPG's. (An efficient algorithm for this purpose was described in Section 4.8.) The resulting CPG's are shown in Figure 9.3.3 and Figure 9.3.4. The striking difference between the two graphs is that although all the processes in Figure 9.3.3 allow independent access implementation, the cycle in Figure 9.3.2 generates a minimal process in Figure 9.3.4 that contains conflicting domains, which therefore does not allow independent access.

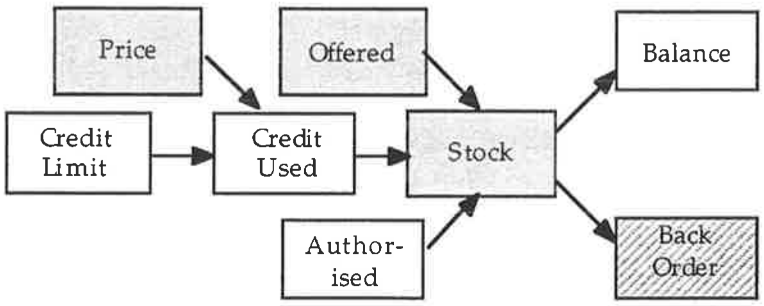

FIGURE 9.3.3: THE CANONICAL PROCESS GRAPH OF FIGURE 9.3.1.

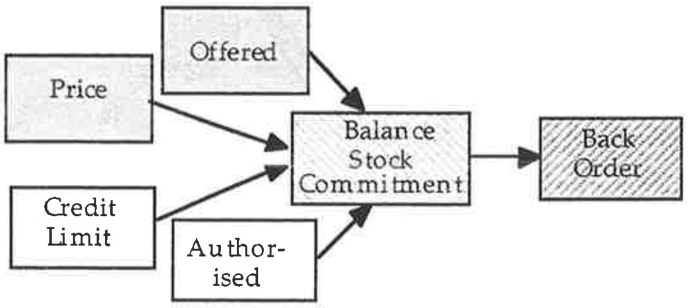

FIGURE 9.3.4: THE CANONICAL PROCESS GRAPH OF FIGURE 9.3.2.

9.4 The Cost Function

To illustrate the optimisation methods, some specific costs for measuring the quality of solutions must be assumed. There are several forms this cost function could take. An obvious choice is to estimate the average execution time for each event. Alternatively, events might have deadlines, some perhaps demanding faster response than others. It is even possible that *parts* of events could have different deadlines, for example where a slow back-end batch system has an interactive front end. Ideally, the cost information would be given as part of the problem specification. This has not been done here, so a simplified cost function will be assumed. In fact, the optimum process graph is often rather insensitive to costs, so a simple function works quite well. The assumptions are these:

1. It costs 10 units for a process to randomly access any set of attributes that share the same index. The cost is considered to be dominated by the cost of searching for the record that has the required index.

2. Independent access causes a speed up of 10, so that accessing a set of attributes independently costs only 1 unit. (It hardly matters how much speed up is associated with independent access, provided it is large compared with the number of processes involved. This ensures that any process graph that uses only independent access is better than any process graph that doesn't.)

3. The cost of transferring delayed procedure calls between processes is negligible. (In any case, since the effect of the first two cost factors is to minimise the number of processes (but not at the cost of independence), they also minimise the cost of the data flows.)

These costs are sufficient to evaluate any proposed implementation — and to demonstrate the properties of the optimisation problem. The canonical process graph of Figure 9.3.3 contains 8 independent access processes, so it has a cost of 8 units. The CPG of Figure 9.3.4 contains one random access process, which accesses two indices, and 5 independent access processes, so it has a cost of 25 units.

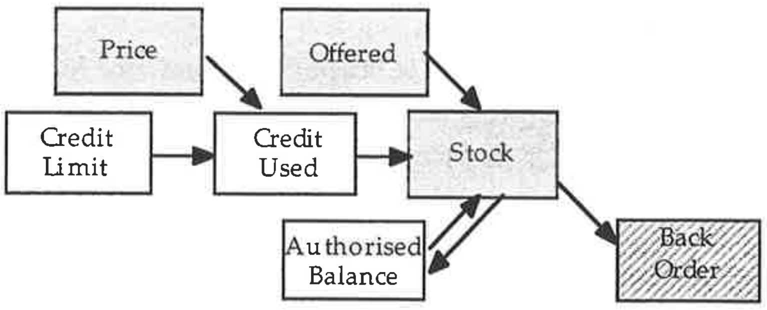

FIGURE 9.4.1: AN INFEASIBLE PROCESS GRAPH

Before considering optimisation algorithms, it is worth optimising these process graphs informally, to highlight the underlying principles. Considering Figure 9.3.3, it pays to combine compatible processes. For example, it pays to combine 'Authorised', 'Credit Limit' and 'Credit Used'. However, it does not pay to combine any of these with 'Balance' because the resulting graph would contain a cycle, like that shown in Figure 9.4.1. Figure 9.4.1 could only be made feasible by then combining 'Stock', 'Balance' and 'Authorised' into a single process. Since the resulting process would access attributes with two unrelated domains, it would no longer be capable of using independent access.

The optimal composite process graph that can be generated from Figure 9.3.3 is shown in Figure 9.4.2. It may be derived by a sequence of three feasible compositions: {Stock} with {Offered}, {Credit Limit} with {Credit Used} and {Authorised} with {Credit Limit, Credit Used}. It has a total cost of 5 units, compared with 8 for Figure 9.3.3. Since there are no feasible compositions between compatible processes remaining in Figure 9.4.2, no more improvement can be made.

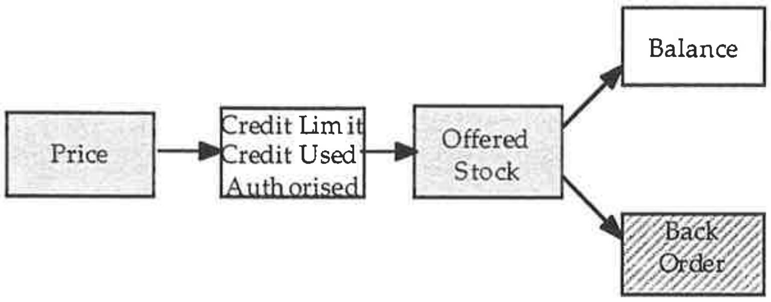

FIGURE 9.4.2: AN OPTIMAL COMPOSITION OF FIGURE 9.3.3.

Now consider the CPG of Figure 9.3.4. It allows two feasible compositions between compatible attributes, of {Price} with {Offered}, and of {Credit Limit} with {Authorised}, as shown in Figure 9.4.3. It has a cost of 23 units (rather than 25).

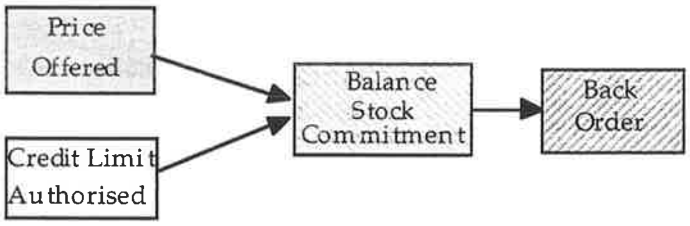

FIGURE 9.4.3: A FEASIBLE COMPOSITION OF FIGURE 9.3.4

It may seem that no further improvement is possible, because any further compositions are sure to destroy independence either in {Price, Offered} or {Credit Limit, Authorised}. But it does not follow that destroying their independence is a loss. Inspection of the original specification (Example 6.6.1) reveals that the set of indices used to access 'Price' and 'Offered' is the same as that used to access 'Stock'. So by placing all three attributes in the same table, the access to 'Stock' retrieves the other two attributes at no extra cost. Likewise, it pays to put 'Credit Limit', 'Authorised', 'Commitment' and 'Balance' into the same table. It does not pay to combine the accesses to 'Back Order' with these, because a different set of rows is involved. In this example, the optimal process graph accesses all the attributes except 'Back Order' within a single process for a total cost of 21 units.

The fact that a canonical process graph shows that attributes are compatible does not necessarily mean that they must have the same set of indices. Agreement is the absence of conflicting indices. It would be possible for the same graph to represent a situation where 'Price' and 'Offered' were accessed by one set of event procedures, and 'Stock' was accessed by a disjoint set. Alternatively, it would be possible that, although the same event accessed 'Offered' and 'Stock', the access to 'Stock' was conditional on the value of 'Offered'. If the condition is rarely true, 'Offered' may be accessed much more often than 'Stock'. If either possibility held, Figure 9.4.3 might be better than the 'optimal' process graph. Unfortunately, there is not enough information in the specification to judge which graph is truly better — and the simple cost function doesn't take account of it anyway.

9.5 A Greedy Heuristic Method

A 'greedy algorithm' is one that seeks to minimise a cost function by making the greatest local savings first. One advantage of greedy methods is that they are fast. Greedy algorithms use one or more local optimisation rules, or 'heuristics', which they continue to apply until no more improvement is possible. There is no consideration of alternatives. Their secondary advantage is that the heuristics may substitute for the cost function, and no costs need be computed — indeed, the true costs need not be known. Four rules are proposed here. They assume a cost function such that independent access is always preferred. However, no greedy algorithm can guarantee to find an optimal solution to every process composition problem.

The heuristics are based on the observation that it always pays to compose two compatible processes — indeed, this is what we mean by saying that two processes are compatible. Unfortunately, some compatible compositions would introduce cycles into the process graph, making it infeasible. Therefore, two compatible processes should be composed only if their composition is feasible. (The rules are similar to an optimising rule in ISDOS [Nunamaker 1971, Nunamaker *et al.* 1976, Nunamaker & Kosynski 1981]).

To discover such pairs of components, it is best to consider the transitive roots of process graphs. (The transitive root of an acyclic graph preserves only the longest paths between vertices.) Candidate processes for composition are either unconnected or adjacent in the graph of the transitive root. Although the CPG is itself a transitive root, a composition may generate a composite process graph with redundant paths, so that its transitive root needs to be recomputed.

Consider the CPG of Figure 9.3.3. It allows five feasible compositions, 'Offered' may be combined with either 'Price' or 'Stock', and any pair chosen from 'Credit Limit', 'Commitment' and 'Authorised' may be combined. Figure 9.5.1 shows the effect of merging 'Credit Limit' with 'Credit Used'. Figure 9.5.2 shows a second composition, in which 'Credit Limit' and 'Credit Used' are merged again with 'Authorised'. Because of the lattice property of feasible process graphs, the order in which 'Credit Limit', 'Credit Used' and 'Authorised' are combined does not affect the outcome.

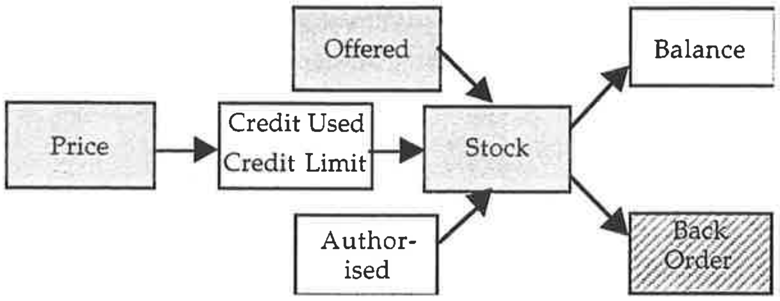

FIGURE 9.5.1: A FEASIBLE COMPOSITION APPLIED TO FIGURE 9.3.3.

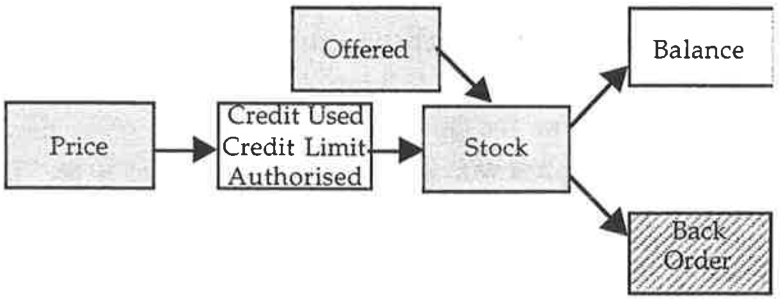

FIGURE 9.5.2: A FEASIBLE COMPOSITION APPLIED TO FIGURE 9.5.1.

A third composition of 'Offered' with 'Price' leads to the process graph of Figure 9.5.3. This graph exhausts the opportunities for merging processes. It has a cost of 5 units, and is an optimal composition. Combining 'Offered' with 'Stock' instead of 'Price' would have led to the alternative optimal process graph of Figure 9.4.2.

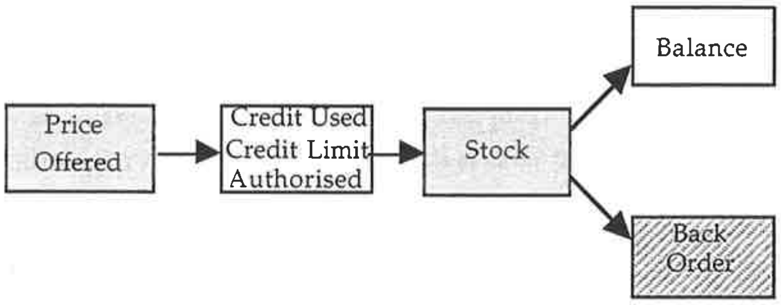

FIGURE 9.5.3: ANOTHER OPTIMAL COMPOSITION OF FIGURE 9.3.3

The decision to use feasible compositions of compatible processes is not in itself sufficient to define an algorithm. It does not state which of several possible compositions should be chosen first, and although the idea of compatibility has been discussed with respect to variables, the compatibility of processes has not yet been defined. (Process compatibility will be discussed in a later section.)

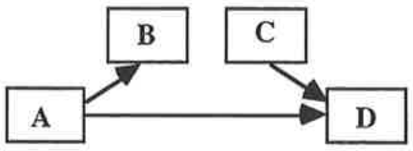

FIGURE 9.5.4: A PROCESS GRAPH OFFERING A CHOICE OF COMPOSITIONS

Given a choice, which feasible composition should be chosen? Figure 9.5.4 shows a process graph that allows two feasible compositions: {B} with {C}, and {A} with {D}. Figure 9.5.5 shows that the effect of composing {B} with {C} is to make the {A,D} composition infeasible. Conversely, Figure 9.5.6 shows that the effect of composing {A} with {D} is to make the {B,C} composition infeasible.

If all the processes concerned allow independent access, both compositions generate a process graph with a cost of 3 units. If none of them do, the cost is 30 units. But what if only {A} and {D} allow independent access? The cost of Figure 9.5.5 is then 12 units and the cost of Figure 9.5.6 is 21 units. Conversely if only {B} and {C} allow independent access, the cost of Figure 9.5.5 is 21 units and the cost of Figure 9.5.6 is 12 units. This example suggests that it is wiser to compose random access processes before independent access ones. Applying this rule systematically to the CPG of Figure 9.3.4, or its composition in Figure 9.4.3, leads to a single process that accesses all the attributes.

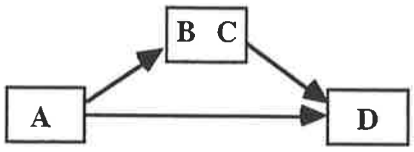

FIGURE 9.5.5: THE EFFECT OF COMPOSING TWO UNCONNECTED PROCESSES

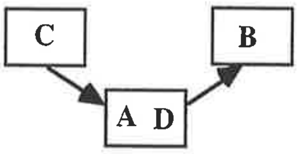

FIGURE 9.5.6: THE EFFECT OF COMPOSING TWO CONNECTED PROCESSES

Other things being equal, is it better to compose two connected processes or two unconnected processes? In Figure 9.5.5, merging {B} and {C} eliminates an access; but in Figure 9.5.6, merging {A} and {D} eliminates an access *and* a data flow. Also, as a rule, unconnected pairs are usually associated with input parameters or with calls to external events. These vertices are not associated with accesses, so composing them achieves nothing. As a rule of thumb, therefore, it pays to compose connected vertices before composing unconnected ones. With luck, the compositions of adjacent vertices will make composing unconnected pairs of vertices unnecessary. (Consider Figure 9.4.3, where each of the independent access processes would be merged into the random access process, by-passing the two unconnected compositions that generated Figure 9.4.3.)

Four heuristic rules are proposed, as follows:

1 Compose random access processes with adjacent compatible processes.

2 Compose pairs of adjacent independent access compatible processes.

3 Compose random access processes with unconnected compatible processes.

4 Compose pairs of unconnected independent access compatible processes.

If more than one rule is applicable, the first rule listed should be applied. If there are several possible applications of the same rule, the choice is arbitrary. The set of rules should be applied until no more applications are possible. It is therefore implicit that conflicting processes are never combined, and that being able to distinguish 'compatible' processes is part of the method. Although these heuristics cannot guarantee to find the optimal solution in the artificial case of Figure 9.6.1 below, they perform well in practice.

9.6 The Complexity of the Optimisation Problem

Process compositions do not commute. Figure 9.6.1 shows a graph that permits three compositions: {C} with {D}, {E} with {B}, and {E} with {F}. Assuming that all the processes allow independent access, the optimal composite process graph is shown in Figure 9.6.2. It is generated by composing {C} with {D}, and {E} with {F} (in either order), and has a cost of 5 units. However, composing {B} with {E} generates the process graph of Figure 9.6.3, which has a cost of 6 units, yet allows no further compositions.

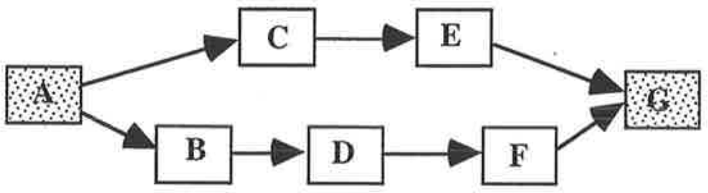

FIGURE 9.6.1: ANOTHER PROCESS GRAPH OFFERING A CHOICE OF COMPOSITIONS

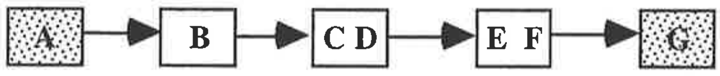

FIGURE 9.6.2: THE OPTIMAL PROCESS GRAPH

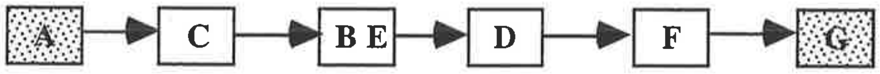

FIGURE 9.6.3: A SUBOPTIMAL PROCESS GRAPH

The time complexity of an algorithm is the function that relates an upper bound of its execution time to a parameter. Usually, the parameter is the length of the encoded input, assuming that the encoding is 'reasonable', i.e., not unnecessarily padded. The exact form of the function is not usually important, and one is more interested in the 'order of complexity' denoted by '$\mathbf{O}(f(n))$', where 'f' is some function of n, the length of the input. For example, to say that the time complexity of an algorithm is $\mathbf{O}(n^2)$, is to say that the actual execution time '$g(n)$' is such that $g(n) \leq kn^2$ for large n and some constant k. (This definition is such that if $g(n)$ is actually $2n$, the complexity may still be correctly described as $\mathbf{O}(n^2)$ (for any $k>0$), although it is better described as $\mathbf{O}(n)$ (with $k \geq 2$). On the other hand, if $g(n)=n^3$, it cannot be described as $\mathbf{O}(n^2)$, because for any finite value of k, there is always a value of n such that $n^3>kn^2$.)

Broadly, algorithms may be classified in two categories according to their complexity: 'tractable' and 'intractable'. Tractable algorithms have complexity $\mathbf{O}(f(n))$, where $f(n)$ is a finite polynomial in n. Since for large n, the highest order term dominates any polynomial, tractable algorithms are often described as having complexity $\mathbf{O}(n^k)$, for some finite $k \geq 0$. However, if there is no finite polynomial that sets an upper bound on its execution time, the algorithm is said to be intractable. Intractable algorithms may have complexities such as $\mathbf{O}(2^n)$, $\mathbf{O}(n!)$, and so on, and are loosely referred to as 'exponentially complex'.

The complexity of a problem is the lowest order complexity of any algorithm that can solve it. Problems are tractable if there are known polynomial algorithms that solve them. Other problems are provably intractable; e.g., finding all permutations of a sequence of length n is intractable because the sequence has $n!$ permutations, so it must take time $\mathbf{O}(n!)$ just to list them.

NP (non-deterministic polynomial) problems lie somewhere between these two categories. They have the property that the correctness of a solution, once it is found, can be checked in polynomial time. However, there are an intractable number of possible solutions. An NP problem could be solved in polynomial time if an oracle could guess the correct solution. NP problems can often be solved by algorithms that are fast *on average*, by using a heuristic in place of the oracle. However, for any given heuristic, there always exist example problems that defeat it, in the sense that either the algorithm fails to find the solution, or it performs no better than an exhaustive search. 'NP-complete' problems are the hardest subclass of NP problems. No polynomial algorithm has ever been found that solves any NP-complete problem (some of which have a long history), nor it has ever been proved that no deterministic polynomial algorithm exists that solves one. If a polynomial algorithm can ever be found to solve any NP-complete problem, then one can be found to solve all NP problems.

Because this thesis offers no polynomial time algorithm that reliably solves the optimal process composition problem, it is obligatory to show it is NP-complete, and therefore it is unlikely that any such algorithm exists. Technically, this is done by showing that a known NP-complete problem can be mapped onto the optimal process composition problem. (Finding a tractable algorithm for the composition problem would then solve the NP-complete problem, and therefore every other NP problem.) In fact, this cannot be proved for all cost functions. For example, if the cost is simply equal to the number of processes, the optimal solution is always a single process, and the problem is trivial. However, it can be proved for some particular cost functions. In particular it can be proved for the combinatorial sub-problem posed in Figure 9.6.1, where the cost is proportional to the number of processes, but only compatible processes may be composed.

The 'Shortest Common Supersequence' problem [Garey & Johnson 1979] is a known NP-complete problem, as follows:

We are given a finite set R of strings from Σ^* over the finite alphabet Σ, and a positive integer K. Is there a string $w \in \Sigma^*$ with $|w| \leq K$ such that each string $x \in R$ is a subsequence of w, i.e., $w = w_0 x_1 w_1 x_2 w_2 \ldots x_k w_k$, where each $w_i \in \Sigma^*$ and $x = x_1 x_2 \ldots$

x_k? This problem is NP-complete provided the alphabet contains at least 5 letters, there are more than 2 strings in R, and each string has more than 2 terms.

Optimising Figure 9.6.1 can be considered as a problem in pairing the matching terms of two sequences. The first sequence is represented by 'C, E', and the second by 'B, D, F', except it is their colours that must be matched, not their names. The alphabet Σ is {white, grey}, and set R is {white-grey, grey-white-grey}. The problem is to find the shortest sequence of colours that contains the original two sequences. Figure 9.6.2 represents the common supersequence 'grey-white-grey', and Figure 9.6.3 represents the common supersequence 'white-grey-white-grey'. Clearly Figure 9.6.2 with 3 terms is the shortest common supersequence, because one of the strings in R also has 3 terms. It is easy to imagine a generalisation of Figure 9.6.1 that has more than 2 sequences of length greater than 2, and at least 5 colours.

Any shortest common supersequence problem can be mapped into a process network like that of Figure 9.6.1 by defining a one-one correspondence from Σ to a set of colours. Each sequence in R is mapped to a corresponding pipeline of processes. Optionally, source and sink processes can be added to complete the correspondence to Figure 9.6.1. It only remains to show that such a process network could result from a system specification. If a typical sequence is $X_1X_2 \ldots X_k$, it is merely necessary to postulate that some event procedure contains the assignment $x_2 := x_1$, some event procedure contains the assignment $x_3 := x_2$, and so on, where for all i from 1 to k, x_i has domain X_i. Therefore, if there exists a polynomial time algorithm to solve the optimal process composition sub-problem, where the cost is proportional to the number of processes, but only compatible processes may be composed, there exists a polynomial time algorithm to solve 'Shortest Common Supersequence'. Therefore at least one version of the process composition problem is NP-complete.

9.7 Branch-and-Bound Search

Although it is unlikely that a tractable algorithm for the optimal composition problem exists, this does not mean that a typical process graph cannot be optimised quickly. An exhaustive search for an optimum would be time consuming if conducted blindly, but a more intelligent search could be expected to succeed in polynomial time in almost all cases — in the sense that although some pathological problems would take more than polynomial time, they would be unlikely to arise in practice, and too rare to affect the average complexity. One such algorithm is 'Branch and Bound Search'.

Choosing whether to make each feasible composition may be treated as a 'design decision'; e.g., given the CPG of Figure 9.3.3 (Page 206), should 'Price' be composed with 'Stock'? Each design decision may be represented by a node in a binary tree. The root of the tree represents a state in which no decisions have been made; its leaves represent states in which all possible decisions have been made, i.e., proposed solutions. For a given problem, there are many possible decision trees, depending on which decisions are considered first.

Figure 9.7.1 shows the first two levels of a possible decision tree for the CPG of Figure 9.3.3. The tree is drawn with its root on the left and its leaves on the right. The complete tree has many leaves. The root corresponds to the question of composing 'Stock' with 'Price'. The upper branch leaving the root represents the decision to combine them, its lower branch represents the decision never to combine them. Decisions are irrevocable. All the solutions in the upper subtree access 'Price' and 'Stock' in the same system component; all the solutions in the lower subtree access them in different components. By inspecting Figure 9.3.3, it is clear that combining 'Price' with 'Stock' implies that their composite would be cyclically connected to the 'Credit Used' process, so in turn the decision implies that all three attributes must be accessed in the same process. In the upper subtree, separating 'Credit Used' from 'Price' or 'Stock' is no longer an option.

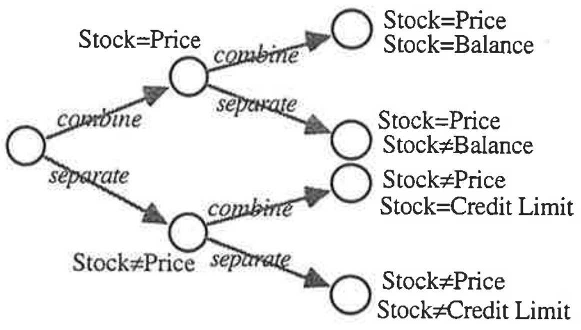

FIGURE 9.7.1: PART OF A DECISION TREE

The second decision depicted in Figure 9.7.1 is whether to combine 'Stock' with 'Credit Limit'. For example, the topmost right hand node represents the case where 'Stock' and 'Price' are combined, and 'Stock' and 'Credit Limit' are combined. It follows that all three attributes must be accessed by the same process (along with 'Credit Used' too), so that it may not later be decided to keep 'Price' and 'Credit Limit' separate. Although it is not shown here, the development of the tree should continue to the right until all possible pairs of compositions have been considered. Logically, wherever a process network allows a choice between random and independent access, both possibilities should be considered. In practice, the choice will almost always favour independent access.

Constructing the complete decision tree would merely be a systematic way of enumerating all possible solutions, of which there are usually too many to consider. (Even the simple process graph of Figure 9.5.2 has 11 valid compositions.) Branch and Bound Search is a way of elaborating only the 'interesting' part of the decision tree, by temporarily ignoring less promising alternatives. This is done by assigning a cost to each node of the decision tree. The cost of each leaf, representing a solution, is its true cost. The costs of the internal nodes, representing partial solutions, have to be estimated. The defining property of Branch and Bound Search is that the cost estimate is always a lower bound; the true cost is always at least the estimated cost. As will be shown shortly, this implies that if a leaf of the decision tree has a lower cost than any other node evaluated by the search, then it must be an optimal solution. The speed of Branch and Bound Search depends on the accuracy with which the estimated costs approximate the true costs and the order in which design decisions are considered.

A lower bound for the cost an internal node of the decision tree may be estimated by assuming optimistically that every set of attributes with a given domain may be accessed in the same process. In the case of Figure 9.3.3 the cost of the root of the decision tree is estimated by assuming that all the 'customer' attributes can be accessed in one independent access process, the 'product' attributes can be accessed in a second independent access process, and 'Back Order' can be accessed in a third independent access process. Therefore the estimated cost is 3 units. It is not certain that only three processes will be needed, so the true cost may be more than 3 units, but it can certainly be no less.

Suppose the first design decision considered is whether to combine 'Credit Limit' and 'Balance' in one process. Consider keeping them separate. This means that there must be at least two processes accessing the 'customer' domain. Therefore the cost is at least 4 units. Now consider combining them. This would create a cycle between the proposed {Credit Limit, Balance} process and the {Stock} process. Consequently, the proposal to compose 'Credit Limit' with 'Balance' is also a proposal to compose them with the other attributes in the resulting cycle, in this case with 'Stock' and 'Credit Used'. The newly formed process would access two different domains, and could not exploit independent access. The proposed process has a cost of 21 units. Including the cost of independent access to 'Back Order', the total lower bound cost of this proposal is 21 units.

The essence of Branch and Bound Search is that the unpromising proposal to compose 'Credit Limit' with 'Balance' is not rejected outright; it is set aside for possible consideration later. Although keeping the two attributes separate looks more promising, it is not certain to lead to any leaf with a cost of less than 21 units. However, since its estimated cost is lower, Branch and Bound Search will consider the proposal to keep 'Credit Limit' and 'Balance' separate in greater depth before it will reconsider the proposal to combine them. Indeed, if it finds a leaf with a cost less than 21 units, it will never reconsider the proposal to combine them. The proposals (i.e., the internal nodes of the decision tree) that have yet to be considered are stored in a priority list, in order by cost. At this stage, the list looks like Table 9.7.1, so that the proposal to keep 'Credit Limit' and 'Balance' separate ought to be considered next.

Cost	**Decisions Made**
4	Credit Limit ≠ Balance.
21	Credit Limit = Balance.

TABLE 9.7.1: OUTCOMES AFTER THE FIRST DECISION POINT

The second decision might be to combine or to separate 'Credit Limit' and 'Authorised'. Keeping them separate (while still keeping 'Credit Limit' and 'Balance' separate) has a cost of 4 units. (The simple-minded method of estimating the lower-bound costs assumes that 'Authorised' and 'Balance' can be composed cheaply later, and fails to foresee that this would create a cycle.) Composing 'Credit Limit' and 'Authorised', while still keeping 'Credit Limit' and 'Balance' separate, also leads to a cost of 4 units. The state of the search is now given by Table 9.7.2.

Cost	Decisions Made
4	Credit Limit ≠ Balance, Credit Limit ≠ Authorised.
4	Credit Limit ≠ Balance, Credit Limit = Authorised.
21	Credit Limit = Balance.

TABLE 9.7.2: OUTCOMES AFTER THE SECOND DECISION POINT

It is arbitrary which of the first two alternatives is explored next. Suppose it is the first one listed. The next decision might be whether to combine 'Price' and 'Stock'. Composing them would create a cycle involving 'Credit Used', and the cost would be at least 21 units. Leaving them separate gives a cost of 4 units. Assuming that decisions of equal cost are taken in first-come-first-served order, the state of the search would then be as given by Table 9.7.3.

Cost	Decisions Made
4	Credit Limit ≠ Balance, Credit Limit = Authorised.
4	Credit Limit ≠ Balance, Credit Limit ≠ Authorised, Price ≠ Stock.
21	Credit Limit = Balance.
21	Credit Limit ≠ Balance, Credit Limit ≠ Authorised, Price = Stock.

TABLE 9.7.3: OUTCOMES AFTER THE THIRD DECISION POINT

Two decisions made so far have shown a large difference in cost between their two outcomes, one has shown no difference. Where there is no difference or a small difference in cost, Branch and Bound Search must pursue several alternative partial solutions. Where the costs differ a lot, the partial solution with the higher cost may not need to be considered again. Any partial solution whose estimated cost exceeds the true cost of the optimal solution will certainly never be reconsidered. It is best to consider first those decisions that make the greatest difference to the cost, thus reducing the number of promising partial solutions to be considered. Therefore, rather than continue the solution above, it is better to examine a search where the most significant decisions are made first.

For this problem, the significant decisions are those that either consider composing two processes with different domains, or that consider composing two processes with the same domain, but whose composition would create a cycle involving a third process with a different domain. There are 23 such significant decisions, where merging the vertices would generate partial solutions with a cost of 21 or more. In whatever order they are considered, they lead to the following partial solution, which needs at least two independent access processes accessing the 'customer' domain and at least two independent access processes accessing the 'product' domain, as shown in Table 9.7.4.

Cost	Decisions Made
4	Back Order ≠ Balance, Back Order ≠ Authorised, Back Order ≠ Credit Used, Back Order ≠ Credit Limit, Back Order ≠ Offered, Back Order ≠ Price, Back Order ≠ Stock, Price ≠ Balance, Price ≠ Authorised, Price ≠ Credit Used, Price ≠ Credit Limit, Offered ≠ Balance, Offered ≠ Authorised, Offered ≠ Credit Used, Offered ≠ Credit Limit, Stock ≠ Balance, Stock ≠ Authorised, Stock ≠ Credit Used, Stock ≠ Credit Limit, Credit Limit ≠ Balance, Authorised ≠ Balance, Credit Used ≠ Balance, Price ≠ Stock.
21	...

TABLE 9.7.4: OUTCOMES AFTER THE 23RD DECISION POINT

The 8 attributes in Figure 9.3.1 form possible 28 pairs, of which 23 have already been considered, leaving the following 5:

Offered = Price?,
Offered = Stock?,
Authorised = Credit Limit?,
Authorised = Credit Used?,
Credit Limit = Credit Used?

These are exactly the possible compositions that would be considered by the heuristic method. It is no coincidence the Branch and Bound Search initially rejects the others.

Taken alone, the first two decisions make no difference to the cost, but deciding against any of the last three would imply a 3rd access of the 'customer' domain, increasing the cost to 5 units. Because a cost differential of one is greater than a cost differential of zero, the last three decisions listed are considered first, giving the partial solution, shown in Table 9.7.5.

Cost	Decisions Made
4	Back Order ≠ Balance, Back Order ≠ Authorised, Back Order ≠ Credit Used, Back Order ≠ Credit Limit, Back Order ≠ Offered, Back Order ≠ Price, Back Order ≠ Stock, Price ≠ Balance, Price ≠ Authorised, Price ≠ Credit Used, Price ≠ Credit Limit, Offered ≠ Balance, Offered ≠ Authorised, Offered ≠ Credit Used, Offered ≠ Credit Limit, Stock ≠ Balance, Stock ≠ Authorised, Stock ≠ Credit Used, Stock ≠ Credit Limit, Credit Limit ≠ Balance, Authorised ≠ Balance, Credit Used ≠ Balance, Price ≠ Stock, Authorised = Credit Limit, Authorised = Credit Used, Credit Limit = Credit Used.
5	...

TABLE 9.7.5: OUTCOMES AFTER THE 26TH DECISION POINT

The remaining two decisions, 'Offered = Price?' and 'Offered = Stock?', still have no cost difference when considered individually. Considering the first, it leads to two partial solutions, both of cost 5, one with 'Offered = Price', the other with 'Offered ≠ Price'. The first choice cannot be combined with the choice 'Offered = Stock' because that would imply 'Price = Stock', a decision that has already been excluded, so that solution is infeasible. However, it may combine with 'Offered ≠ Stock' to give a solution of cost 5 units. Since no decisions remain to be made, it is a complete solution. The list of partial solutions contains no proposal with an estimated cost less than 5 units, so the solution is optimal. This important property of Branch and Bound Search, that the first solution found is optimal, holds only because the cost estimates of partial solutions are lower bounds of their true costs.

There are actually two optimal solutions. The solution described is shown in Table 9.7.6, which corresponds to the process graph shown in Figure 9.5.1. If the search were continued, the remaining partial solution of cost 5 would lead to the alternative shown in Table 9.7.7, which differs from the first only in its last two terms, and corresponds to the process graph of Figure 9.4.2 (Page 208). Once the classic Branch and Bound Search has found one leaf of the decision tree it will not search for another. However, the classic search procedure can easily be modified to yield alternative solutions, which it will generate in order of increasing cost.

Cost	Decisions Made
5	Back Order ≠ Balance, Back Order ≠ Authorised, Back Order ≠ Credit Used, Back Order ≠ Credit Limit, Back Order ≠ Offered, Back Order ≠ Price, Back Order ≠ Stock, Price ≠ Balance, Price ≠ Authorised, Price ≠ Credit Used, Price ≠ Credit Limit, Offered ≠ Balance, Offered ≠ Authorised, Offered ≠ Credit Used, Offered ≠ Credit Limit, Stock ≠ Balance, Stock ≠ Authorised, Stock ≠ Credit Used, Stock ≠ Credit Limit, Credit Limit ≠ Balance, Authorised ≠ Balance, Credit Used ≠ Balance, Price ≠ Stock, Authorised = Credit Limit, Authorised = Credit Used, Credit Limit = Credit Used, Offered = Price, Offered ≠ Stock.

TABLE 9.7.6: 1ST SOLUTION FOUND BY BRANCH-AND-BOUND

Cost	Decisions Made
5	Back Order ≠ Balance, Back Order ≠ Authorised, Back Order ≠ Credit Used, Back Order ≠ Credit Limit, Back Order ≠ Offered, Back Order ≠ Price, Back Order ≠ Stock, Price ≠ Balance, Price ≠ Authorised, Price ≠ Credit Used, Price ≠ Credit Limit, Offered ≠ Balance, Offered ≠ Authorised, Offered ≠ Credit Used, Offered ≠ Credit Limit, Stock ≠ Balance, Stock ≠ Authorised, Stock ≠ Credit Used, Stock ≠ Credit Limit, Credit Limit ≠ Balance, Authorised ≠ Balance, Credit Used ≠ Balance, Price ≠ Stock, Authorised = Credit Limit, Authorised = Credit Used, Credit Limit = Credit Used, Offered ≠ Price, Offered = Stock.

TABLE 9.7.7: 2ND SOLUTION FOUND BY BRANCH-AND-BOUND

From the example, it may be seen that by taking significant decisions first, Branch and Bound Search may need no more steps to find an optimal solution than there are pairs of vertices in the CPG. However, its effectiveness depends critically on having a good estimate of the cost of partial solutions. As an example of a very poor cost function, a cost can never be less than 0, therefore 0 is a valid lower bound estimate for every partial solution. However, the function would fail to differentiate between good and bad proposals, and since no leaf has a zero cost, the search would generate every possible solution, of which there are very many. The ideal estimator would be the true cost, but finding it would be time-consuming, since it can only be found by search. The trick is to find an estimate that is as close to the true cost as possible, but which is easy to compute.

A second example shows what happens when the implementation must use random access. Consider the CPG of Figure 9.3.4. It has 15 pairs of vertices, so 15 design decisions must be made. The lower bound on the cost at the root of the decision tree is 21 units, which assumes that although 'Back Order' can be accessed independently, random access is needed to only two different index values. This is because all the 'customer' or 'product' index values involved are equal.

As before, the decisions with the highest cost difference should be considered first. These are the decisions to keep any of the original processes separate from one other, each of which increases the cost by at least 1 unit. In whichever order these alternatives are considered, they will all be rejected. Therefore all the vertices should belong to the same component process, and the cost of the solution is 21 units.

9.8 Hybrid Search

It is reasonable to ask whether its is possible for an algorithm to combine the best aspects of Branch and Bound Search with the greedy heuristic method. This is encouraged by the observation that, provided the most significant decisions are taken first, Branch and Bound Search begins by rejecting infeasible compositions, testing only the same compositions considered by the greedy method. Bearing in mind that any valid process graph can be reached purely by feasible compositions, the infeasible compositions will stay rejected until a solution is found. Therefore Branch and Bound Search only needs to consider the same set of compositions as the greedy method. In the case of the CPG of Figure 9.3.3, the number of decisions would be reduced from 28 to 5. This vastly reduces the size of the search space. The whole decision tree would be similar to the subtree generated by the partial solution given in Table 9.7.4.

It is interesting to see how such a hybrid method copes with the combinatorial problem presented by the process network of Figure 9.6.1. The actual cost of this graph is 7 units. Its lower bound cost is 3 units, because it involves 3 different domains. There are 3 feasible compositions: {C} with {D}, {B} with {E}, and {E} with {F}. These all reduce the cost by 1 unit, and all 3 are unconnected compositions, so there is nothing to choose between them. Suppose that {B,E} is considered first. It generates two sub-solutions given in Table 9.8.1. Since the {B,E} composition allows no further feasible compositions, its cost is known to be 6 exactly. The decision to keep {B} and {E} separate still has a lower bound of 4 units, because there must now be at least 4 processes.

Cost	Decisions Made
4	B≠E
6	B=E

TABLE 9.8.1: AFTER THE 1ST DECISION

With {B} and {E} separate, there is still nothing to distinguish the composition {C,D} from the composition {E,F}. Suppose the composition {C,D} is evaluated first. Combining {C} and {D} has a cost of at least 4, and separating then has a cost of at least 5. The state of the search is then given by Table 9.8.2.

Cost	Decisions Made
4	B≠E, C=D
5	B≠E, C≠D
6	B=E

TABLE 9.8.2: AFTER THE 2ND DECISION

The only possible composition remaining is {E,F}. Making it has an actual cost of 5, whereas leaving {E} and {F} separate has an actual cost of 6. Assuming that alternatives of equal cost are consider in first-come-first-served order, the state of the search is then given by Table 9.8.3.

Cost	Decisions Made
5	B≠E, C≠D
5	B≠E, C=D, E=F
6	B≠E, C=D, E≠F
6	B=E

TABLE 9.8.3: AFTER THE 3RD DECISION

Separating {B} and {E}, and separating {C} and {D} still allows the {E,F} composition, after which the heuristic allows no further feasible compositions. The alternatives are shown in Table 9.8.4. Since the lowest cost item allows no further optimisation, it is a solution. Since no other solution can cost less, it is the optimal solution.

Cost	Decisions Made
5	B≠E, C=D, E=F
6	B≠E, C=D, E≠F
6	B=E
6	B≠E, C≠D, E=F
7	B≠E, C≠D, E≠F

TABLE 9.8.4: AFTER THE 4TH DECISION

In this example the hybrid method found an optimum solution after considering only 4 decisions and 3 possible compositions, whereas the pure Branch and Bound Search would have to consider 21 possible compositions. This seems to suggest that the hybrid method is clearly superior. The truth is not so clear cut. Assuming that the pure method considers the most significant decisions first, it will reject (i.e. defer forever) the 18 infeasible compositions because of their high cost, leaving only the feasible compositions. After rejecting the infeasible decisions the pure search will then proceed in the same way as the hybrid search. The main benefit of the heuristic is that composing only adjacent and unconnected vertices may prove to be a faster way of avoiding cycles than composing first and testing later. Also, the heuristic method reduces the size of the graph by one vertex after every composition. On the other hand, the Branch and Bound method considers all pairs of vertices, and the hybrid method considers all feasible compositions. Therefore, the greedy heuristic method can be expected to need less iterations than the other two methods.

9.9 Process Compatibility

The notion of process compatibility was introduced as the basis of the greedy heuristic method. By definition, two processes are compatible if combining them would reduce the cost of the process network. Conversely, the greedy method assumes that combining two compatible processes will reduce the cost of the process network. As a result, the precise definition of process compatibility gives fine control over the results of the greedy method. In addition, apart from its formal use in the greedy method, process compatibility is a useful aid to commonsense reasoning. As explained in Chapter 6, commonsense reasoning may help a human system designer adjust the system specification so that it has an efficient implementation. What rules may a designer use to decide whether two processes should be combined?

Process compatibility is based on the same ideas as compatibility of variables introduced in Section 5.4. To a first approximation, if two processes access compatible variables, the two processes are compatible. This means that two independent access processes with the same domain may be composed providing that (1) their composition is 'feasible', and (2) there are no dependences between variables with different index definitions. However, this simple rule cannot deal with the case where one or other process already can't use independent access. The notion of many-one or 'partial' agreement further clouds this simple picture.

Processes can be divided into two broad categories, independent and random access. It would never be wise to merge two independent access processes if their composite was a random access process. On the other hand, combining them is beneficial if the variables they access are compatible. This allows their variables to be placed in the same table, replacing two accesses by one. Even if the sets of indices they access are disjoint, their composition will not increase execution time appreciably (accessing larger records takes only slightly longer), and it will certainly save file space. Likewise, when two processes both use random access, combining them is always harmless, and, if any index is accessed by both of them, the combined process will replace two accesses by one.

It is possible to have various degrees of independence. Referring back to the example of Example 5.1.1, there is a possibility of independence by 'class' or by '(class, student)'. Clearly there are more instances of '(class, student)' than of 'class' alone. Therefore, it is better to have '(class, student)' independence than 'class' independence, and it would be unwise to combine a doubly independent '(class, student)' process with a singly independent 'class' process. In practice, their composition may prove harmless in a parallel implementation because there are not enough physical processors to exploit the extra parallelism, i.e., there may be far fewer processors than 'classes'. On the other hand, in a sequential access implementation, combining the two would mean that that the '(class, student)' domain had to be accessed randomly, albeit clustered by 'class'. How we should treat many-one compatibility therefore depends on the intended form of the implementation. In the case of composite domains we should add a footnote to the cost function of Section 9.4. When a process is independent we assign it a speed up of 10, so when a process is doubly independent we should assign it a speed up of 100, and so on, otherwise the cost function will treat many-one compatibility just like one-one compatibility .

Finally, there is compatibility of a kind between an independent and a random access process if the independent access process accesses a subset of the rows accessed by the random access process. (This was discussed in connection with the optimisation of Figure 9.4.3.) The point is that if the random access process must access all the rows accessed by the independent access process, it may as well retrieve the variables the independent access process needs as well. It is easy to decide if one set of indices is a subset of another, although the textual use of an index does not always imply its use during execution. It may be that the access is conditional, as in '**if** A(i) = 0 **then** B(i) := false; **end if**;'. The accesses to 'B' are a subset of the accesses to 'A', but the accesses to 'A' are not likely to be a subset of the accesses to 'B'. Thus, if 'A' was accessed in a independent access process and 'B' in a random access process, then it would be unwise to combine them, but if 'A' was accessed in a random access process and 'B' in a independent access process, then it would be wise. This matter can be resolved by reference to the text of the specification. However, it can also be approximated by reference to the SDG, because the state of 'B' depends on the state of 'A'. Therefore, it may be unwise to combine the processes if the independent access process is upstream of the random access process. What should be done if the processes are unconnected? It is hard to tell from the SDG alone. An unconnected pair of 'A' and 'B' vertices could be generated by '**if** C(i)=0 **then** A(i):=0; **else** B(i):=0; **end if**;' in which case their accesses are disjoint, and they certainly should not be combined, or by '**begin** A(i):=C(i); B(i):=C(i); **end**;' in which case their accesses are the same, and they should be combined. Since an independent access process has negligible cost compared with a random access process, there is little benefit in ever combining them, and sometimes a great risk. However, in the case that all the variables of the independent access process depend on variables of the random access process with the same indices, the composition is always safe and beneficial. This may be detected in the SDG by noting that each independent access process variable has a compatible dependence on a random access process variable.

The above remarks may be generalised to a pair of independent access processes whose domains have a many-one relationship, e.g., '(class, student)' and 'class', by replacing references to 'the random access process' by 'the process with the lesser degree of independence' and to 'the independent access process' by 'the process with the greater degree of independence'. Therefore, the following two heuristics define 'process compatibility':

1 Two processes agree if they have the same degree of independence, and all their variables are compatible at that level of independence.

2 An independent access process agrees with a process of lesser independence if the indices accessed by the more independent process are a subset of the indices accessed by the less independent process, and all the shared indices in the more independent process depend on indices in the first process.

Variables local to event procedures do not cause special difficulty unless they have been reused, either as discussed in Section 4.5, or by being used within a loop. Where a local variable is reused within a loop, it should be declared within the loop, as in Example 2.6.2, and may notionally be replaced by an array of variables. A variable used to accumulate the result of a reduction operation (discussed in Section 2.6.4) is a special case. An accumulator variable, needed in the specification language to model reduction, *must* be reused. It actually is reused in a sequential access implementation, but something more subtle than simple reuse occurs in a parallel access one. It is important to access an accumulator variable in the same process as the loop that controls the reduction, otherwise the $\mathbf{O}(\log n)$ reduction tree would be replaced by a more costly $\mathbf{O}(n)$ set of delayed procedure calls to a separate process. Its association with the correct process can be forced by treating the accumulator *as if* it had one instance for each iteration of the loop body, typically making it agree with the variable or expression it accumulates. On the other hand, outside the loop, it must be considered to have one instance for the loop as a whole. In the sense that an accumulator reduces a set of values to a single value, it must present two different faces. This is difficult to show on an SDG, and no special notation is used to show it.

Because of the potentially complex nature of process compatibility, it is impractical to fully label composite process graphs with compatibility information. This is not very satisfactory for a human designer equipped with a process graph, who must continually refer to the text of the specification to decide what optimisations are possible. A better approach is for the designer to work directly with the CPG, enclosing proposed composite processes within outlines. Figure 9.9.1 shows how outlines (the grey ellipses) can be drawn on Figure 9.3.1 to correspond to the optimal process graph of Figure 9.4.2.

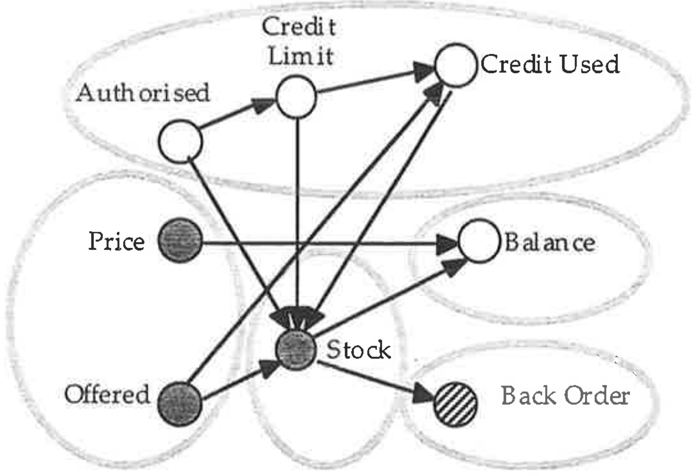

FIGURE 9.9.1: A PARTITIONED SDG

Since they represent composite processes, the designer must draw the outlines so that there is no cycle between them. Outlines that contain compatible vertices can be implemented as independent access processes; outlines that contain incompatible vertices must be implemented as random access processes. Provided the CPG is not too large, an informal approach to choosing the best outlines works surprisingly well in practice.

The needs of the *Designer* CASE tool are somewhat different from those of a human designer. A human designer works best with a simple graph, but can apply sophisticated reasoning. Indeed, SDG's for human use typically show one vertex per variable, and only the state variables are shown. On the other hand, the graphs generated by use-definition analysis contain at least one vertex for each lexical definition of every variable. They are typically much more complex, and very difficult for a human to comprehend. In particular, they introduce a further problem in optimisation, which is that simple local variables can be many-one compatible with more than one state variable. It is as if simple local variables are uncoloured, and can assume whatever colour is desired — but only one. The number of possible compositions is therefore much greater than it is for a graph showing only state variables. It is also easy to show that the problem remains NP-complete; the case when all the vertices are coloured is NP-complete, and this case is a subset of the case where some vertices may be initially uncoloured. Exactly how this additional complication is handled is explained in Chapter 10.

10. The Designer CASE Tool

This chapter describes a CASE tool, *Designer*, that can design an efficient batch processing system to implement a given specification. The tool is implemented as a program written in Prolog [Clocksin & Mellish 1984, O'Keefe 1990], and listed in the Appendix (Chapter 13). Prolog was chosen as the implementation language for several reasons: it provides a non-deterministic grammar notation that simplifies the construction of the parser, it provides pattern matching on data structures, which can be used to manipulate syntax trees, it has library routines for operations on sets and directed graphs, and its pattern matching and non-determinism make it an ideal language for writing heuristic rule-based systems. Similar reasons for the choice of Prolog as an implementation language for an optimising Pascal compiler can be found in [Gabber 1990].

Briefly, the method is as follows: The system specification is parsed, then the resulting abstract syntax tree is analysed to discover the dependences between definitions. The SDG that is constructed is a form of use-definition graph in which each vertex represents a definition of a variable, and each edge represents a use of one definition in constructing another. The use-definition analysis needed is more complex than is normal (e.g., in a compiler), in order to keep track of compatibility information. Once the SDG has been constructed, it is analysed to discover its strongly-connected components. Each strong component corresponds to a separable process. The reduced component graph, which is always acyclic, links these separable processes by first-in-first-out data flows, and is a feasible system topology. This process graph is canonical, and derives uniquely from the specification. Unfortunately, it contains the greatest possible number of processes, and is not as efficient as it might be. The CPG is optimised by successively combining pairs of processes, provided that doing so does not reduce the degree of independent access that is possible. When no more compatible pairs can be found, the design is complete. It is then known what processes the system should contain and what variables they access, so it is possible in principle to generate the procedures that each process should contain. As was seen in Chapter 7, generating them is tedious but straightforward.

The *Designer* CASE tool is engineered as a 'proof of concept', and is not a particularly robust program. However, it can correctly analyse all the specifications given in the earlier chapters. The optimiser uses the greedy heuristic method, which finds an optimum solution in each case.

The program consists of five major phases:

1 The *Parser* converts a textual specification into an abstract syntax tree.

2 The *Analyser* converts the abstract syntax tree into an SDG.

3 The *Canoniser* converts the SDG into a CPG.

4 The *Optimiser* simplifies and optimises the process graph.

5. The *Generator* allocates the nodes of the abstract syntax tree to processes.

The *Generator* phase is not the *Programmer* tool described in Chapter 7. It does not generate process specifications, but merely provides the process labels assumed at the start of Chapter 7, e.g., in Example 7.1.2. The five phases can be called as an integrated system, or each can be run separately, reading and writing transfer files. The advantage of being able to run the phases separately is to aid debugging.

10.1 Formal Syntax

To be acceptable to the the *Designer* program, system specifications must be stated in a *subset* of the specification language. Specifically, there is only one package: the system itself. In addition, although *Designer* allows internal functions, it does not allow internal procedures. Any internal procedures must therefore be expanded in-line. This restriction is imposed to simplify the *Designer* program, but it is not a restriction on what can be specified. The syntax of the language accepted by *Designer* is given in Table 10.1.1.

There are also some restrictions on the use of variable names that are not present in Ada: The names of variables may not be overloaded. There can only be one variable in scope with a given name. If a local variable has the same name as a global variable, the global variable becomes invisible. (This is because the *Designer* program is not sophisticated enough to resolve which of several variables might be meant by a given name.) In addition, the names of loop variables must not duplicate the names of local variables, or one another.

system → *packages generics*
package *identifier* **is**
  event_specification { *event_specification* }
end *identifier* **;**
package body *identifier* **is**
  { *variable_declaration* } { *function* } *event* { *event* }
end identifier **;**

packages → { **with** *identifier* **;** }

generics → [**generic** *generic_type* {*generic_type* }]

generic_type → **type** *identifier* **is private ;** | **type** *identifier* **is range <> ;**

identifier_list → *identifier* { **,** *identifier* }

type_declaration → [**array** (*identifier_list*) **of**] *identifier*

event_specification → **procedure** *identifier* [(*parameters*)] **;**

event → **procedure** *identifier* [(*parameters*)] **is**
  { *variable_declaration* }
begin *statements* **end** *identifier* **;**

function → **function** *identifier* (*parameters*) **return** *identifier* **is**
  { *local_variable_declaration* }
begin *statements* **end** *identifier* **;**

parameters → *parameter_declaration* { **;** *parameter_declaration* }

parameter_declaration → *identifier_list* **:** *type_declaration*

variable_declaration → *identifier_list* **:** *type_declaration* [*initialiser*] **;**

statements → *statement* { *statement* }

statement → **null ;**

statement → **return** [*expression*] **;**

statement → **if** *expression* **then** *statements* [*elsif_part*] **end if ;**

elsif_part → **elsif** *expression* **then** *statements* [*elsif_part*] | **else** *statements*

statement → **while** *expression* **loop** *statements* **end loop ;**

statement → **all** *identifier* **in** *identifier* **loop** *statements* **end loop ;**

statement → **for** *identifier* **in** *identifier* **loop** *statements* **end loop ;**

statement → *identifier* **.** identifier [(*expression_list*)] **;**

statement → **declare** { *local_variable_declaration* } **begin** *statements* **end ;**

statement → *variable* **:=** *expression* **;**

variable → *identifier* [(*identifier* { **,** *identifier* })]

expression → *term* { *infix_operator term* }

term → *prefix_operator term* | (*expression*) | *identifier* (*expression_list*)
  | *constant* | *variable*

expression_list → *expression* { **,** *expression* }

initialiser → **:=** *constant* | **:=** *mapping*

mapping → (**others =>** *mapping*) | (**others =>** *constant*)

infix_operator → **+** | **-** | ***** | **/** | **mod** | **&** | **or** | **and** | **=** | **/=** | **<=** | **<** | **=>** | **>**

prefix_operator → **+** | **-** | **not**

TABLE 10.1.1: SYNTAX OF THE SPECIFICATION LANGUAGE

10.2 The Parser

The purpose of the *Parser* is to convert a system specification into an abstract syntax tree. The syntax of the language accepted by the parser was given in Section 10.1. Example 10.2.1 shows the specification of the Macrotopian Reference Library system, as presented to the parser. Example 10.2.2 shows the syntax tree the parser created as a result.

```
with report;
generic
 type title is private;
 type user is private;
package Library is
 procedure borrow (t : title; u : user);
 procedure reshelve (t : title; u : user);
 procedure buy (t : title);
 procedure audit;
end Library;
package body Library is
 C : array (title) of natural := (others => 0);
 D : array (user) of natural := (others => 0);
 procedure borrow (t : title; u : user) is
 begin
  if C(t) > 1 then
   C(t) := C(t)-1;
   D(u) := D(u)+1;
  end if;
 end borrow;
 procedure reshelve (t : title; u : user) is
 begin
  C(t) := C(t)+1;
  D(u) := D(u)-1;
 end reshelve;
 procedure buy (t : title) is
 begin
  C(t) := C(t)+1;
 end buy;
 procedure audit is
  stock, loans : natural := 0;
 begin
  all t in title loop
   stock := stock + C(t);
  end loop;
  all u in user loop
   loans := loans + D(u);
  end loop;
  report.audit(stock, loans);
 end audit;
end Library;
```

EXAMPLE 10.2.1: SPECIFYING THE LIBRARY SYSTEM

```
system(library,
 packages([report]),
 generics([title,user]),
 [state(lex([global/library/d,user],1),natural),
  state(lex([global/library/c,title],1),natural)],
 [event(borrow,
   [param(lex([input/borrow/u],1),user),
    param(lex([input/borrow/t],1),title)],
   [],
   [if(lex([internal/borrow/expn_1],1),
     [lex([global/library/c,input/borrow/t],2)],
     [assign(lex([global/library/c,input/borrow/t],3),
       [lex([global/library/c,input/borrow/t],4)]),
      assign(lex([global/library/d,input/borrow/u],2),
       [lex([global/library/d,input/borrow/u],3)])],[])]),
  event(reshelve,
   [param(lex([input/reshelve/u],1),user),
    param(lex([input/reshelve/t],1),title)],
   [],
   [assign(lex([global/library/c,input/reshelve/t],5),
     [lex([global/library/c,input/reshelve/t],6)]),
    assign(lex([global/library/d,input/reshelve/u],4),
     [lex([global/library/d,input/reshelve/u],5)])]),
  event(buy,
   [param(lex([input/buy/t],1),title)],
   [],
   [assign(lex([global/library/c,input/buy/t],7),
     [lex([global/library/c,input/buy/t],8)])]),
  event(audit,
   [],
   [local(lex([local/audit/loans],1),natural),
    local(lex([local/audit/stock],1),natural)],
   [all(lex([loop/audit/t],1),
     title,
     [assign(lex([local/audit/stock],2),
       [lex([local/audit/stock],3),
        lex([global/library/c,loop/audit/t],9)])]),
    all(lex([loop/audit/u],1),
     user,
     [assign(lex([local/audit/loans],2),
       [lex([local/audit/loans],3),
        lex([global/library/d,loop/audit/u],6)])]),
    call(lex([output/report/audit],1),
     [lex([local/audit/stock],4),
      lex([local/audit/loans],4)])])]).
```

EXAMPLE 10.2.2: THE SYNTAX TREE OF EXAMPLE 10.2.1.

The *Parser* generates no output from event declarations; only their implementations within the package body are represented in the abstract syntax tree. The immediate children of its root are these: the generic package declarations, the generic type declarations, a list of state variable declarations, and a list of event definitions. The subtree for each event definition comprises three parts: the declarations of its parameters, the declarations of its local variables, and a list of the statements in the procedure body.

The *Parser* detects syntax errors, but its diagnostics and error-recovery are poor. It also detects cases where a variable is undeclared. Unlike in Ada, only one definition of a name can be in scope. For example, the parser does not allow a loop variable to have the same name as another loop variable, parameter or local variable within the same event. Its type checking is rudimentary. It checks only that array elements have appropriate indices. For example, if the domain of 'Back_Order' is '(customer, product)', in the reference to the element 'Back_Order(Who, What)' it verifies that 'Who' has type 'customer' and 'What' has type 'product'. The aim of the program is demonstrate feasibility; it is not a consumer product.

10.2.1 Variables

Each variable is characterised by its domain and codomain. For example, the declaration:

C : **array** (title) **of** natural;

has domain 'title' and codomain 'natural'. If the variable is a state variable, the parser extends the name of the variable into the form '**global/***system/variable*', where *system* is the name of the system, and *variable* is the name of the variable, in lower-case characters. For example, if the system name is 'library', 'C' would be represented internally as 'global/library/c'. The complete subtree for the declaration of 'C' would be 'state(lex([global/library/c,title],1), natural)'. (See Example 10.2.2.) Extending the identifier serves two purposes: to distinguish the variable from any local variables with the same name, and to identify references to the variable instantly as references to a state variable.

The names of parameters and local variables are extended like those for state variables. Parameter names are extended to '**input/***event/variable*' and local variables are extended to '**local/***event/variable*', where *event* is the name of the event. Variables declared by **for** and **all** loops are extended to '**loop/***event/variable*', and dummy variables used to represent **if** and **while** statement control expressions are extended to '**internal/***event/variable*'. Thus all distinct variables are given distinct names.

Each occurrence of a variable in the text is formed into a lexical definition of the form 'lex(*variable*,*n*)', where *variable* is the extended identifier of the variable, and *n* is unique to each occurrence of the variable. In addition, a dummy variable is associated with each control expression occurring in an **if** or **while** statement (e.g., 'expn_1' in Example 10.2.2). This variable serves two purposes: its eventual allocation to a process determines the allocation of its control statement, and its use usually reduces the number of edges in the SDG.

10.2.2 Statements

The body of an event definition comprises a '**begin** ... **end**' list of statements, which is represented internally as a list of subtrees. Each statement in the list is represented by a tree whose root labels the type of statement: 'assign', 'call', 'null', 'return', 'if', 'for', 'all' or 'while'. (See Example 10.2.2.) The number of children varies according to the type of node. Since **if**, **for**, **all** and **while** statements enclose statement lists, some of their children are also lists of subtrees. The parser simplifies single-branch **if** statements and multi-branch **if** statements containing **elsif** clauses into nested two-branch **if** statements — possibly with empty statement lists. Other possible children of statement nodes are event names, variables, variable declarations, and expressions. Variable declarations that arise from **for** and **all** statements have already been discussed, and are treated in exactly the same way as local variables. Event names, which appear in procedure calls, are extended similarly to variable names, to the form '**event/***package/event*', where *package* is the name of the called package, and *event* is the name of the called event procedure.

Variables are also represented as lists. The first element of such a list is the extended name of the variable itself, which is then followed by the names of any subscripts it may have. For example, a reference to 'C(t)' within the 'Borrow' event is represented as '[global/library/c, input/borrow/t]'. (*See* Example 10.2.2.) A list representation does not allow nested subscripts or subscript expressions, but the specification language only permits subscripts to be simple variables anyway.

An identifier is expanded to its extended name by inspecting a two-level symbol table. The outer level lists the names of system variables; the inner level lists the names of parameters and local variables for the event currently being analysed. In the case that a name refers to both a local variable (or parameter) and to a state variable, the local variable is chosen. When loop variables are encountered, or variables are declared within blocks, e.g., within an **all** loop, the inner level of the symbol table is extended, but it is not contracted again when the variable passes out of scope. Therefore, all local variables must have distinct names. This is not a limitation in principle. It would be possible to have multiple declarations with the same name. The parser would then have to give synonyms serial numbers, in order to distinguish them. However, it is already confusing enough to distinguish the different lexical definitions and dynamic definitions of the same variable, so this extra complication was foregone. Furthermore, it deals neatly with problem of loop folding dealt with in Section 7.5. Loops with different loop variables are considered incompatible, so unnested loops cannot be considered compatible, and loop folding cannot occur.

Expressions are represented as lists of variable representations; constants are not included. Since only dependences matter, the parser suppresses the internal structure of expressions. Thus, the list '[lex([local/e/x],3),lex([local/e/y],2)]' might represent the expression 'x–y+1', the expression 'min(x,y)', or many others. The internal workings of functions are irrelevant to dependence analysis, so they are ignored by the parser. This has the advantage that the specifier need not say how to implement a complex function. For example, if 'income_tax' is calculated from 'taxable_income' in some way, it is sufficient to write 'income_tax := tax(taxable_income)'; it is unnecessary to specify the implementation of the 'tax' function.

10.3 The Dependence Analyser

The dependence analyser derives an SDG from the abstract syntax tree created by the parser. The principles underlying dependence analysis were discussed in Chapter 8. Each definition within the program text may have several dynamic definitions. Dynamic definitions are represented as records of the form 'dd(*variable*, *flag*, *frequency*)'. *Variable* is a list of triples. In the case of a simple variable, the list contains a single triple. In the case of a reference to an array element, the list contains a triple for the array name, followed by a triple for each of its indices. Each triple consists of an extended variable name, its lexical definition number, and its dynamic definition number. Dynamic definitions can refer to entire arrays, in which case their indices are replaced by triples of the form '(*,0,0)'.

Lexical definition numbers uniquely identify the places in the specification that define the variable and its indices. Dynamic definition numbers, in conduction with the variable name, uniquely identify the dynamic definition itself. *Flag* is either 'rd', which is used when a definition inspects an array element, or 'wr', which is used when a definition is created by an assignment. Finally, *frequency* is a list representing the loop structure enclosing the point where the dynamic definition is made, and measures the number of times the definition can be made within a single call of the event procedure in which it appears. Each enclosing **for** or **all** loop is represented by the triple for its loop variable, and a **while** loop is represented by the triple for a dummy variable that is assigned the value of the control expression. The frequency information is used by the optimiser to determine compatibility.

Example 10.3.1 shows part of the output from the *Analyser*. It represents the SDG in the form of two incidence matrices. The first matrix shows the 'hard' edges, and the second shows the 'soft' edges. Each row represents a dynamic definition. To read the output, consider any given row. Each '^' or 'v' entry indicates a dependence or 'use'. The reader's eye should follow the 'v' down, or the '^' up, vertically, until reaching a row containing 'o' or '@'. This row contains a definition on which the given definition depends. The 'o' entries simply mark the diagonal of the matrix. An entry of '@' shows a loop, i.e., a definition that depends directly on itself. The remaining entries: '.', '_', and '|' are merely to help the eye.

Given the assignment 'x:=y;', for example, the use-definition graph will contain an edge *from* a definition of 'x' *to* a definition of 'y'. The direction chosen for the internal representation of an edge is *the reverse of that used in the diagrams* in previous chapters: from 'x' to 'y', indicating that 'x *uses* y', rather than 'y *is used by* x'. This is because the *Analyser* asks the question 'What definitions are used by x?' more often than 'What definitions use y?'

```
ovv.|....|....|....|....|....|....|....| dd([(global/library/c,0,4),(input/borrow/t,1,1)],wr,[])
.o..|....|....|v...|....|....|....|....| dd([(global/library/c,2,2),(input/borrow/t,1,1)],rd,[])
.^o.|....|....|v...|v...|....|....|....| dd([(global/library/c,3,3),(input/borrow/t,1,1)],wr,[])
...ov....|....|...v|....|....|....|....| dd([(global/library/c,5,7),(input/reshelve/t,1,1)],wr,[])
____o____|____|___v|____|____|____|____| dd([(global/library/c,6,6),(input/reshelve/t,1,1)],rd,[])
....|ov..|....|..v.|....|....|....|....| dd([(global/library/c,7,10),(input/buy/t,1,1)],wr,[])
....|.o..|....|..v.|....|....|....|....| dd([(global/library/c,8,9),(input/buy/t,1,1)],rd,[])
....|..o.|....|....|....|....|v...|....|
                                     dd([(global/library/c,9,12),(loop/audit/t,1,2)],rd,[(loop/audit/t,1,1)])
....|...o|....|....|....|....|..v.|....|
                                     dd([(global/library/c,9,14),(loop/audit/t,1,4)],rd,[(loop/audit/t,1,1)])
____|____ov___|_v__|v___|____|____|____| dd([(global/library/d,2,3),(input/borrow/u,1,1)],wr,[])
....|....|o...|.v..|....|....|....|....| dd([(global/library/d,3,2),(input/borrow/u,1,1)],rd,[])
....|....|.ov.|....v....|....|....|....| dd([(global/library/d,4,6),(input/reshelve/u,1,1)],wr,[])
....|....|..o.|....v....|....|....|....| dd([(global/library/d,5,5),(input/reshelve/u,1,1)],rd,[])
....|....|...o|....|....|....|....|v...|
                                     dd([(global/library/d,6,8),(loop/audit/u,1,2)],rd,[(loop/audit/u,1,1)])
____|____|____o____|____|____|____|__v_|
                                     dd([(global/library/d,6,10),(loop/audit/u,1,4)],rd,[(loop/audit/u,1,1)])
....|....|....|o...|....|....|....|....| dd([(input/borrow/t,1,1)],rd,[])
....|....|....|.o..|....|....|....|....| dd([(input/borrow/u,1,1)],rd,[])
....|....|....|..o.|....|....|....|....| dd([(input/buy/t,1,1)],rd,[])
....|....|....|...o|....|....|....|....| dd([(input/reshelve/t,1,1)],rd,[])
____|____|____|____o____|____|____|____| dd([(input/reshelve/u,1,1)],rd,[])
.^..|....|....|....|o...|....|....|....| dd([(internal/borrow/expn_1,1,1)],wr,[])
....|....|....|....|.ov.v....|....|....| dd([(local/audit/loans,0,4)],wr,[(loop/audit/u,1,1)])
....|....|....|....|..o.|....|....|....| dd([(local/audit/loans,1,1)],rd,[])
....|....|...^|....|..^ov....|....v....| dd([(local/audit/loans,2,2)],wr,[(loop/audit/u,1,1)])
____|____|____^____|___^o____|____|v___| dd([(local/audit/loans,2,3)],wr,[(loop/audit/u,1,1)])
....|....|....|....|....|ov.v|....|....| dd([(local/audit/stock,0,4)],wr,[(loop/audit/t,1,1)])
....|....|....|....|....|.o..|....|....| dd([(local/audit/stock,1,1)],rd,[])
....|..^.|....|....|....|.^ovv....|....| dd([(local/audit/stock,2,2)],wr,[(loop/audit/t,1,1)])
....|...^|....|....|....|..^o|.v..|....| dd([(local/audit/stock,2,3)],wr,[(loop/audit/t,1,1)])
____|____|____|____|____|____o____|____| dd([(loop/audit/t,1,1)],wr,[(loop/audit/t,1,1)])
....|....|....|....|....|....^o.v.|....| dd([(loop/audit/t,1,2)],wr,[(loop/audit/t,1,1)])
....|....|....|....|....|....^.o..|....| dd([(loop/audit/t,1,3)],wr,[(loop/audit/t,1,1)])
....|....|....|....|....|....|.^o.|....| dd([(loop/audit/t,1,4)],wr,[(loop/audit/t,1,1)])
....|....|....|....|....|....^..^o|....| dd([(loop/audit/t,1,5)],wr,[(loop/audit/t,1,1)])
____|____|____|____|____|____|____o____| dd([(loop/audit/u,1,1)],wr,[(loop/audit/u,1,1)])
....|....|....|....|....|....|....^o.v.| dd([(loop/audit/u,1,2)],wr,[(loop/audit/u,1,1)])
....|....|....|....|....|....|....^.o..| dd([(loop/audit/u,1,3)],wr,[(loop/audit/u,1,1)])
....|....|....|....|....|....|....|.^o.| dd([(loop/audit/u,1,4)],wr,[(loop/audit/u,1,1)])
....|....|....|....|....|....|....^..^o| dd([(loop/audit/u,1,5)],wr,[(loop/audit/u,1,1)])
____|____|____|____|_^__|^___|____|____o dd([(output/report/audit,1,1)],wr,[])

o...|....|....|....|....|.. dd([(global/library/c,0,4),(input/borrow/t,1,1)],wr,[])
.o..|v...|....|....|....|.. dd([(global/library/c,1,1),(*,0,0)],rd,[])
^^o.|....|....|....|....|.. dd([(global/library/c,1,5),(*,0,0)],wr,[])
.^.o|....v....|....|....|.. dd([(global/library/c,1,8),(*,0,0)],wr,[])
_^__o____|v___|____|____|__ dd([(global/library/c,1,11),(*,0,0)],wr,[])
.^..|o...|...v|....|....|.. dd([(global/library/c,1,13),(*,0,0)],wr,[(loop/audit/t,1,1)])
.^..|^o..|....|....|....|.. dd([(global/library/c,1,15),(*,0,0)],wr,[(loop/audit/t,1,1)])
....|.^o.|....v....|....|.. dd([(global/library/c,1,16),(*,0,0)],wr,[(loop/audit/t,1,1)])
.^..|...o|....|....|....|.. dd([(global/library/c,2,2),(input/borrow/t,1,1)],rd,[])
____|____o____|____|____|__ dd([(global/library/c,5,7),(input/reshelve/t,1,1)],wr,[])
.^..|....|o...|....|....|.. dd([(global/library/c,6,6),(input/reshelve/t,1,1)],rd,[])
....|....|.o..|....|....|.. dd([(global/library/c,7,10),(input/buy/t,1,1)],wr,[])
.^..|....|..o.|....|....|.. dd([(global/library/c,8,9),(input/buy/t,1,1)],rd,[])
.^..|....|...o|....|....|.. dd([(global/library/c,9,12),(loop/audit/t,1,2)],rd,[(loop/audit/t,1,1)])
____|^___|____o____|____|__ dd([(global/library/c,9,14),(loop/audit/t,1,4)],rd,[(loop/audit/t,1,1)])
....|....|....|o..v|....|.. dd([(global/library/d,1,1),(*,0,0)],rd,[])
....|....|....|^o..|.v..|.. dd([(global/library/d,1,4),(*,0,0)],wr,[])
....|....|....|^.o.|...v|.. dd([(global/library/d,1,7),(*,0,0)],wr,[])
....|....|....|^..o|....|v. dd([(global/library/d,1,9),(*,0,0)],wr,[(loop/audit/u,1,1)])
____|____|____|^__^o____|__ dd([(global/library/d,1,11),(*,0,0)],wr,[(loop/audit/u,1,1)])
....|....|....|....^o...|.v dd([(global/library/d,1,12),(*,0,0)],wr,[(loop/audit/u,1,1)])
....|....|....|....|.o..|.. dd([(global/library/d,2,3),(input/borrow/u,1,1)],wr,[])
....|....|....|^...|..o.|.. dd([(global/library/d,3,2),(input/borrow/u,1,1)],rd,[])
....|....|....|....|...o|.. dd([(global/library/d,4,6),(input/reshelve/u,1,1)],wr,[])
____|____|____|^___|____o__ dd([(global/library/d,5,5),(input/reshelve/u,1,1)],rd,[])
....|....|....|^...|....|o. dd([(global/library/d,6,8),(loop/audit/u,1,2)],rd,[(loop/audit/u,1,1)])
....|....|....|...^|....|.o dd([(global/library/d,6,10),(loop/audit/u,1,4)],rd,[(loop/audit/u,1,1)])
```

EXAMPLE 10.3.1: PART OF THE OUTPUT FROM THE ANALYSER

```
[(lex([global/library/d,user],1),natural),
 (lex([global/library/c,title],1),natural),
 (lex([input/borrow/u],1),user),
 (lex([input/borrow/t],1),title),
 (lex([input/reshelve/u],1),user),
 (lex([input/reshelve/t],1),title),
 (lex([input/buy/t],1),title),
 (lex([local/audit/loans],1),natural),
 (lex([local/audit/stock],1),natural),
 (lex([loop/audit/t],1),title),
 (lex([loop/audit/u],1),user)].

[dd([(global/library/c,1,1),(*,0,0)],rd,[]),
 dd([(global/library/c,1,5),(*,0,0)],wr,[]),
 dd([(global/library/c,1,8),(*,0,0)],wr,[]),
 dd([(global/library/c,1,11),(*,0,0)],wr,[]),
 dd([(global/library/c,1,16),(*,0,0)],wr,[(loop/audit/t,1,1)]),
 dd([(global/library/d,1,1),(*,0,0)],rd,[]),
 dd([(global/library/d,1,4),(*,0,0)],wr,[]),
 dd([(global/library/d,1,7),(*,0,0)],wr,[]),
 dd([(global/library/d,1,12),(*,0,0)],wr,[(loop/audit/u,1,1)]),
 dd([(output/report/audit,1,1)],wr,[])].

[dd([(global/library/c,0,4),(input/borrow/t,1,1)],wr,[])-
  [dd([(global/library/c,2,2),(input/borrow/t,1,1)],rd,[]),
   dd([(global/library/c,3,3),(input/borrow/t,1,1)],wr,[])],
 dd([(global/library/c,2,2),(input/borrow/t,1,1)],rd,[])-
  [dd([(input/borrow/t,1,1)],rd,[])],
 dd([(global/library/c,3,3),(input/borrow/t,1,1)],wr,[])-
  [dd([(global/library/c,2,2),(input/borrow/t,1,1)],rd,[]),
   dd([(input/borrow/t,1,1)],rd,[]),
   dd([(internal/borrow/expn_1,1,1)],wr,[])],
 dd([(global/library/c,5,7),(input/reshelve/t,1,1)],wr,[])-
  [dd([(global/library/c,6,6),(input/reshelve/t,1,1)],rd,[]),
   dd([(input/reshelve/t,1,1)],rd,[])],
 dd([(global/library/c,6,6),(input/reshelve/t,1,1)],rd,[])-
  [dd([(input/reshelve/t,1,1)],rd,[])],
 dd([(global/library/c,7,10),(input/buy/t,1,1)],wr,[])-
  [dd([(global/library/c,8,9),(input/buy/t,1,1)],rd,[]),
   dd([(input/buy/t,1,1)],rd,[])],
 dd([(global/library/c,8,9),(input/buy/t,1,1)],rd,[])-
  [dd([(input/buy/t,1,1)],rd,[])],
 dd([(global/library/c,9,12),(loop/audit/t,1,2)],rd,[(loop/audit/t,1,1)])-
  [dd([(loop/audit/t,1,2)],wr,[(loop/audit/t,1,1)])],
 dd([(global/library/c,9,14),(loop/audit/t,1,4)],rd,[(loop/audit/t,1,1)])-
  [dd([(loop/audit/t,1,4)],wr,[(loop/audit/t,1,1)])],
 dd([(global/library/d,2,3),(input/borrow/u,1,1)],wr,[])-
  [dd([(global/library/d,3,2),(input/borrow/u,1,1)],rd,[]),
   dd([(input/borrow/u,1,1)],rd,[]),
   dd([(internal/borrow/expn_1,1,1)],wr,[])],
 dd([(global/library/d,3,2),(input/borrow/u,1,1)],rd,[])-
  [dd([(input/borrow/u,1,1)],rd,[])],
 dd([(global/library/d,4,6),(input/reshelve/u,1,1)],wr,[])-
  [dd([(global/library/d,5,5),(input/reshelve/u,1,1)],rd,[]),
   dd([(input/reshelve/u,1,1)],rd,[])],
 dd([(global/library/d,5,5),(input/reshelve/u,1,1)],rd,[])-
  [dd([(input/reshelve/u,1,1)],rd,[])],
 dd([(global/library/d,6,8),(loop/audit/u,1,2)],rd,[(loop/audit/u,1,1)])-
  [dd([(loop/audit/u,1,2)],wr,[(loop/audit/u,1,1)])],
 dd([(global/library/d,6,10),(loop/audit/u,1,4)],rd,[(loop/audit/u,1,1)])-
  [dd([(loop/audit/u,1,4)],wr,[(loop/audit/u,1,1)])],
 dd([(input/borrow/t,1,1)],rd,[])-[],
 dd([(input/borrow/u,1,1)],rd,[])-[],

... and so on ...
```

EXAMPLE 10.3.2: THE INPUT TO THE CANONISER

The analysis algorithm has already been discussed in Chapter 8. The *Analyser* phase uses the single definition method. In the case of a definition that results from merging the branches of an **if** statement, where the definition has no corresponding lexical occurrence, the lexical occurrence number is zero.

Actually, the part of the *Analyser* output shown in Example 10.3.1 is meant for human consumption only. The hard and soft edges are actually transmitted to the *Canoniser* phase as two lists of dynamic definitions, each member of which is paired with the list of definitions on which it depends. This representation is less concise than the matrix form, but is easier for the *Canoniser* to parse, and corresponds directly to its internal representation. These lists are preceded by a list of all the variable declarations occurring in the specification, and a list of all its final definitions, i.e., those that correspond to outputs or the final definitions of state variables. The declarations are not needed by the *Canoniser*, and are simply passed forward to the *Optimiser*. The beginning of the *Analyser* output is shown in Example 10.3.2. (The listing of the SDG is incomplete.) The *Canoniser* ignores the dependence matrix shown in Example 10.3.1. Internally, the *Analyser* stores the dependence information in a third form: a pair comprising a list of vertices and a list of edges, i.e., ordered pairs of vertices.

As explained in Chapter 8, the *Analyser* builds the SDG by traversing the abstract syntax tree prepared by the *Parser*. A typical Prolog predicate within the *Analyser* has the form:

```
analyse_stmt(Context,Statement,DefsIn,DefsOut,Hard,Soft,Decls)
```

where 'Context' is the conditional context, 'Statement' is the subtree whose root is the statement being analysed, 'DefsIn' is the set of live definitions entering the statement, and 'DefsOut' is the set of live definitions leaving the statement. 'Hard' and 'Soft' are the parts of the hard and soft SDG's constructed to model the action of the statement. The SDG's are built bottom up, starting from the leaves of the syntax tree (which represent uses or definitions of variables), the subgraphs being formed into larger graphs using the constructions explained in Chapter 8. Finally, 'Decls' is the list of all internal variable declarations made within the statement. At the top levels of the syntax tree, this list of declarations is concatenated with the lists of parameter, local and state variable declarations, and passed to the *Optimiser* by way of the *Canoniser*.

The conditional context is initially empty, but within the context of an **if** statement or **while** statement it becomes a list containing the dummy variable that represents the control condition. Since the assignment to this variable is made inside the context of any enclosing **if** or **while** statements, the definition uses the existing conditional context, so one term is sufficient.

10.4 The Canoniser

The hard and soft SDG's constructed by the *Analyser* must be combined to give the graph needed for determining separability. The strongly-connected components of this graph correspond to separable processes. It is necessary to ensure that all the dynamic definitions that derive from the same lexical definition are assigned to the same process, otherwise it would be impossible to generate component specifications using the rewriting rules. Also, all the definitions of a state variables must be assigned the same process, as required by the constraint of real-time equivalence. No such treatment is needed for local variables — including those representing elements of state variable arrays.

For these reasons, the *Canoniser* temporarily links together all sets of dynamic definitions that share the same lexical definition, i.e., those with the same variable and lexical occurrence numbers. This forces them into the same strong component, and therefore into the same process. This serves both goals, because the *Analyser* gives all the definitions of state variable arrays the same lexical occurrence number — the value '1', corresponding to the point of their declaration in the specification.

The *Canoniser* constructs the reduced graph of this augmented SDG. There is an edge between a pair of vertices in the reduced graph if there is an edge in the SDG between any pair of members in the components the vertices represent. The *Canoniser* finds the reduced graph using Tarjan's Algorithm [Tarjan 1972], during a single depth-first search of the graph, as described in Section 4.8.

The *Canoniser* then finds the 'transitive root' of the reduced graph. A 'transitive reduction' of a given graph is any subgraph of the graph that has the same transitive closure as the given graph. A 'minimal transitive reduction' of a given graph is a transitive reduction such that none of its proper subgraphs is also a transitive reduction of the given graph. An acyclic graph has a unique minimal transitive reduction, called its transitive root. If one is interested only in the transitive closure properties of a graph, as we are here, its transitive root is a more compact way to represent it. Using transitive roots also simplifies the *Optimiser*.

One way to find a transitive root would be to first find the transitive closure of the SDG, a graph containing an edge for every path of positive length in the graph. Using a related algorithm due to Eve & Kurki-Suonio [Eve & Kurki-Suonio 1997], also described in Section 4.8, the transitive closure of a graph can also be found during one depth-first search. Then, by forming the product of the transitive closure with the original graph, a graph is formed that has an edge corresponding to every compound path of the original graph, i.e., every path of length 2 or more. By subtracting the edges of this product from the original graph, all edges that correspond to compositions of other edges are eliminated [Aho *et al.* 1972]. However, this approach proved too memory-intensive. Instead, each edge of the graph was individually tested to see if it corresponded to a compound path, and was removed if it did. Although testing for the presence of a compound path needed a separate depth-first search from each vertex in turn, rather than the single search needed by Tarjan's Algorithm, this method proved fast enough. In fact, since finding the transitive closure of a graph involves constructing many more edges than are present in the original graph, and the chosen method avoids this, it may actually prove faster in some cases.

The transitive root of the reduced SDG is a feasible system design, i.e., its CPG (canonical process graph).

The output of the *Canoniser* consists of five parts:

- The list of declarations passed to it by the *Analyser*.
- The SDG passed to it by the *Analyser*.

- A list of the strongly connected components of the SDG. (Each component is simply a list of dynamic definitions.)
- The CPG.
- The incidence matrix of the CPG, in which each dynamic definition is mapped onto its corresponding lexical definition.

```
o.v.vv...|...v|.  [lex([global/library/c,*],1),
                   lex([global/library/c,input/borrow/t],0),
                   lex([global/library/c,input/borrow/t],2),
                   lex([global/library/c,input/borrow/t],3),
                   lex([global/library/c,input/buy/t],7),
                   lex([global/library/c,input/buy/t],8),
                   lex([global/library/c,input/reshelve/t],5),
                   lex([global/library/c,input/reshelve/t],6),
                   lex([global/library/c,loop/audit/t],9),
                   lex([internal/borrow/expn_1],1)]
^o.v|.v..|....v.  [lex([global/library/d,*],1),
                   lex([global/library/d,input/borrow/u],2),
                   lex([global/library/d,input/borrow/u],3),
                   lex([global/library/d,input/reshelve/u],4),
                   lex([global/library/d,input/reshelve/u],5),
                   lex([global/library/d,loop/audit/u],6)]
..o.|....|....|.  [lex([input/borrow/t],1)]
...o|....|....|.  [lex([input/borrow/u],1)]
___o_____|____|_  [lex([input/buy/t],1)]
....|o...|....|.  [lex([input/reshelve/t],1)]
....|.o..|....|.  [lex([input/reshelve/u],1)]
....|..o.v....|.  [lex([local/audit/loans],0)]
....|...o|....|.  [lex([local/audit/loans],1)]
_^__|___^o____|_  [lex([local/audit/loans],2)]
....|....|o.v.|.  [lex([local/audit/stock],0)]
....|....|.o..|.  [lex([local/audit/stock],1)]
^...|....|.^o.|.  [lex([local/audit/stock],2)]
....|....|...o|.  [lex([loop/audit/t],1)]
____|____|____o_  [lex([loop/audit/u],1)]
....|..^.|^...|o  [lex([output/report/audit],1)]
```

EXAMPLE 10.4.1: PART OF THE CANONISER OUTPUT

The incidence matrix produced for the Macrotopian Library system is shown in Example 10.4.1. Again, the matrix is to aid debugging, and is not read by the *Optimiser*. It uses the same conventions as Example 10.3.1, in that each row represents a vertex of the graph. However, a vertex now corresponds to a set of dynamic definitions. There are often several dynamic definitions of each lexical definition, but all of them must fall in the same strongly-connected component. Therefore, to make it more concise and more readable, the incidence matrix shows their corresponding lexical definitions.

10.5 The Optimiser

The inputs to the *Optimiser* are the first four outputs produced by the *Canoniser*; the incidence matrix is ignored. In turn, it produces three outputs: a topologically ordered list of processes, the process graph on which the topological ordering is based, and the incidence matrix of the process graph. The incidence matrix for the Macrotopian Reference Library process graph is shown in Example 10.5.1. It has two rows, representing its 'C' and 'D' processes. A process vertex is a pair of values: its degree of independence, and the list of lexical definitions allocated to it. The degree of independence is a list of the domains over which parallel or sequential access is possible. If the list is empty, the process must use single-thread random access. The 'C' process allows independent access by 'title', and the 'D' process allows independent access by 'user'. The matrix indicates that data flows from the 'C' process to the 'D' process

```
o. ([title],[lex([global/library/c,*],1),
     lex([global/library/c,input/borrow/t],0),
     lex([global/library/c,input/borrow/t],2),
     lex([global/library/c,input/borrow/t],3),
     lex([global/library/c,input/buy/t],7),
     lex([global/library/c,input/buy/t],8),
     lex([global/library/c,input/reshelve/t],5),
     lex([global/library/c,input/reshelve/t],6),
     lex([global/library/c,loop/audit/t],9),
     lex([input/borrow/t],1),
     lex([input/buy/t],1),
     lex([input/reshelve/t],1),
     lex([internal/borrow/expn_1],1),
     lex([local/audit/stock],0),
     lex([local/audit/stock],1),
     lex([local/audit/stock],2),
     lex([loop/audit/t],1)])
^o ([user],[lex([global/library/d,*],1),
     lex([global/library/d,input/borrow/u],2),
     lex([global/library/d,input/borrow/u],3),
     lex([global/library/d,input/reshelve/u],4),
     lex([global/library/d,input/reshelve/u],5),
     lex([global/library/d,loop/audit/u],6),
     lex([input/borrow/u],1),
     lex([input/reshelve/u],1),
     lex([local/audit/loans],0),
     lex([local/audit/loans],1),
     lex([local/audit/loans],2),
     lex([loop/audit/u],1),
     lex([output/report/audit],1)])
```

EXAMPLE 10.5.1: PART OF THE OPTIMISER OUTPUT

Some optimisation strategies were discussed in Chapter 9. The *Optimiser* uses the greedy heuristic method, merging vertices in pairs. Merging proceeds in such a way that the graph connecting the merged vertices always remains acyclic, and it is simplified to its transitive root after each merge operation. Therefore the input and output of each optimisation stage are both transitive roots. The *Optimiser* merges pairs of processes until its heuristics can find no more pairs to merge. The resulting graph, which is acyclic, is the finished system design. The merging process is monotonic, and no backtracking occurs. As a result, the finished system design may not be truly optimal, although the *Optimiser* actually finds an optimal design for the examples in this thesis.

10.5.1 The Degree of Independence

Each process is given a degree of independence (or simply, 'degree'), represented as in Example 10.5.1. In general, a degree is a list of domain types, possibly empty. A process also contains a set of definitions. The degree of a process depends only on the definitions that are allocated to it, and the 'hard' edges of the dependence subgraph that connect them. It does not depend on any edges leaving or entering the process or the vertices they connect to. In implementation terms, this reflects the fact that sorting event messages between processes resolves their incompatibilities.

A degree of independence must not only be established for the minimal processes in the CPG, but must continually be re-evaluated for new compositions proposed during optimisation. It is clear that composing a process that is independent with respect to 'customer' with a process that is independent with respect to 'product' will produce a single-thread process. Therefore, the known independence of processes can be used to filter out useless compositions. However, it is not clear that composing two processes that are independent with respect to 'customer' will result in a process that is also independent with respect to 'customer'. This requires detailed investigation to discover whether there is, joining the two processes, a dependence that links definitions with different indices.

A definition has a degree that is determined by its relationship to the definitions it 'uses' in the SDG. The degree of a *definition* is the degree of independent access possible in assigning values to the variable concerned. The degree of a *process* is the common prefix of the degrees of all the definitions it contains.

The degree of a definition has three components: First, it has an 'inherent' degree, which depends on the domain types of the defined variable, but also depends on whether the variable is a state variable or a local variable. Second, it has a degree determined by the definitions it uses. If their definitions are incompatible with the definition's inherent degree, its effective degree must be reduced. Third, its degree may need to be reduced a second time because the loop structure of the event procedure clashes with its degree.

The inherent degree of a state variable is the list of its domain types. For example, the declaration:

Back_Order: **array** (customer, product) **of** natural;

means that the variable 'Back_Order(Who, What)' has degree '[customer, product]' because, in itself, it can be accessed independently by both 'customer' and 'product'.

Local variables and parameters must be treated differently from state variables. Consider an assignment of the form:

Back_Order(Who, What) := Shortage;

where 'Back Order' and 'Shortage' are both allocated to the same process. If 'Shortage' is a state variable, no independence is possible; its degree is empty, because all accesses to 'Shortage' must occur in real-time order. On the other hand, if 'Shortage' is a local variable, then the degree of the assignment may yet be '[customer, product]'. This is because there is a separate instance of 'Shortage' for each instance of the event procedure. The degree of a parameter or local variable is therefore said to be 'extensible'. The *Optimiser* represents an extensible degree by terminating its list of domains with an asterisk. If 'Shortage' is a local variable, its degree is represented as '[*]'. To assess the common degree of two lists, the optimiser treats '*' as a wild-card, so that the common degree of '[customer, product]' and '[*]' is '[customer, product]'.

However, to assess the common degree of an assignment, it is necessary to look at index *definitions*, not domains. For example, if the following sequence is allocated to a single process:

```
t := A(i);
i := j;
B(i) := t;
```

it has degree '[]', because the definition of 'i' used in 'B(i)' is not the same definition used in 'A(i)'. Therefore, the *Optimiser* must sometimes consider degrees that are lists of index definitions rather than lists of domain types. It also follows that definitions cannot be considered in isolation. The degrees of the first and third assignments taken separately are the domains of 'A' and 'B', but their joint degree is empty. Therefore, their degrees depend on whether or not they are assigned to the same process.

The *Optimiser* modifies the degree of each definition by considering all the definitions it uses, either directly or indirectly, provided they lie within the same process. For the process under consideration, it first finds the subgraph of the 'hard' SDG whose vertices correspond to definitions assigned to the process. It then finds the transitive closure of this graph, in which each definition is linked to all the definitions it uses either directly or indirectly.

For each edge of the closure, the *Optimiser* finds the longest common prefix of the index lists of the source and target definitions. This list is then immediately converted to a corresponding list of domain types. The degree of a definition is the common prefix of all the degrees computed in this way, for each edge entering it. (The index lists of local variables are considered extensible, as before.)

In the case of the sequence above, a depth-first search would discover that the definition of 'B(i)' uses definitions of 't' and 'A(i)'. If 'i'' and 'i''' denote the two different definitions of 'i', the *Optimiser* would find the common degree of the lists '[i']', '[*]', to be '[i']', but the common degree of '[i']' and '[i'']' to be empty. After converting these results to domains, their common prefix is empty.

It is important to convert indices to domains for each dependence separately. For example, if 'S' is a local variable in the following context:

```
all i in index loop
   S := S + A(i);
end loop;
```

'S' is compatible with every instance of 'A(i)', even though the instances of 'A(i)' are incompatible with one another. The dependences of 'S' on 'A(i)' are '[i']', '[i'']', and so on, which have an empty common prefix. However, converted to domains, the dependences are '[index]', '[index]', and so on, having the common prefix '[index]'.

Since every assignment made in a process needs to have at least the same degree as the number of independent instances of the process, then conversely, the degree of a process is the longest prefix common to the degrees of all the definitions it contains.

This approach deals correctly with the event:

```
procedure Transfer (Payer, Payee : account; Amount : money) is
begin
   Balance (Payer) := Balance (Payer) – Amount;
   Balance (Payee) := Balance (Payee) + Amount;
end Transfer;
```

There are two lexical definitions of ‘Balance (Payer)’ here. The second depends on ‘Balance’ and ‘Payer’; the first depends on the second, and on the definition of ‘Amount’. Its degree is the common prefix of ‘[Payer]’ with ‘[*]’, and ‘[Payer]’ with ‘[Payer]’, i.e., both yield ‘[Payer]’, which then translate into the domain type ‘[account]’. The degree of the definition is therefore the common prefix of ‘[account]’ and ‘[account]’, i.e., ‘[account]’. Likewise, the degree of the second definition of ‘Balance (Payee)’ is ‘[Payee]’, which is also translated into ‘[account]’. Since every definition in the event has degree ‘[account]’ or ‘[*]’, the degree of the whole ‘Transfer’ procedure is ‘[account]’. More accurately, the degree of the ‘Transfer’ event itself is never established; it is the process associated with ‘Balance’ that has degree ‘[account]’.

Only the dependences *within* a process count. Consider the event:

```
procedure Transfer_2 (Payer, Payee : account; Amount : money) is
begin
   if Authorised (Payer) and Authorised (Payee) then
      Balance (Payer) := Balance (Payer) – Amount;
      Balance (Payee) := Balance (Payee) + Amount;
   end if;
end Transfer_2;
```

If ‘Authorised’ and ‘Balance’ are assigned to separate processes, each process has degree ‘[account]’. If they are assigned to the same process, it will have degree ‘[]’. The dependences of ‘Balance (Payer)’ on ‘Authorised (Payee)’ and of ‘Balance (Payee)’ on ‘Authorised (Payer)’ are only relevant when they share the same process. The *Optimiser* considers only the subgraph of the SDG that includes the vertices assigned to the process concerned. If ‘Authorised’ and ‘Balance’ share the same process, the edges representing the dependence of ‘Balance (Payer)’ on ‘Authorised (Payee)’ and of ‘Balance (Payee)’ on ‘Authorised (Payer)’ fall within the same process’s subgraph, but if they are in different processes, the edges span between the processes, and don’t form part of either subgraph. This is how the *Optimiser* determines that combining the ‘Balance’ and ‘Authorised’ processes would be unproductive, even though they have the same degree.

10.5.2 Preserving Structure

Recalling that the transformation of a system specification into component specifications is based solely on text replacement using delayed procedure call, what aspect of the *Designer* ensures that the transformation is possible? Essentially, it needs to ensure that if Process P calls an event procedure within Process Q, the text of Q’s event procedure can be nested within the text of the calling procedure in Process P. For example, if P contains an **if** statement controlling an assignment in Q, that is acceptable; but it should never create a design where an **if** statement in Q attempts to a control an assignment in P.

This function is taken care of by the use of the conditional context by the *Analyser*, which forces the assignment in Q to depend on the control expression in P. Since loops are treated analogously to **if** statements, no difficulties can arise with loops, either.

A related problem is to preserve the nesting order of **if** statements and loops. Section 8.8 explained that the conditional context forces the nesting of **if** statements to be preserved. Since the *Analyser* models loops similarly to **if** statements, it also forces the nesting order of loops to be preserved. Despite this, it cannot guarantee that the structures assigned to a process are consistent with a sequential or parallel update algorithm. This problem was discussed in Section 7.3, where it was also pointed out that an event procedure may be called many times within the processing of a single event.

Every definition has a 'frequency', which measures the number of times it must be executed. The *Analyser* establishes the frequencies, which it represents as ordered lists. An **all** loop is represented by its loop variable, and a **while** loop by its control expression variable. If the implementation will use parallel access, **for** loops are treated like **while** loops; for sequential access they are treated similar to **all** loops. The *Analyser* currently gives them frequencies like those for **all** loops, which means that it solves for sequential access designs. If a parallel access design is required, the specifier must replace **for** loops by equivalent **while** loops.

The frequency of a definition within an event procedure has two components: the frequency with which the event procedure containing it is called, and the frequency with which the definition is executed within it. The calling frequency and execution frequency can be distinguished by testing to see whether each loop variable or control expression is defined within the same process as the definition in question. If it is, the loop must be part of the same process; if it isn't, the loop must belong to the calling process. The list of loop variables in the calling process therefore defines the calling frequency.

In the calling frequency is deleted from the frequency of a definition, the residue is the definition's execution frequency within the called procedure. But the execution frequency and degree of parallelism of the process containing it must be compatible, otherwise the procedure does not have a loop structure capable of fitting into an independent update algorithm. To check this, the *Optimiser* must infer the loop structure of each independent process. In particular, independent access can only support a loop structure that is a simple nesting. For example, if an event has the form:

```
all i in index loop
   all j in domain loop
      ...
   end loop;
   all k in key loop
      ...
   end loop;
end loop;
```

the independent update algorithm's loop structure cannot be '[i,j]', because that leaves no place to embed the statements with the loop on 'k', nor can it be '[i,k]', because of the statements in the loop on 'j'. It is only possible to achieve independence with respect to 'i', and the 'j' and 'k' loops cannot give rise to extra independence. This gives rise to the idea that 'j' and 'k' should be called 'invalid' indices, i.e., they cannot contribute to independence. The degree of independence is '[i]' in this case, not because it is the common prefix of '[i,j]' and '[i,k]', but because 'i' does not conflict with any other loop, and is 'valid'. Further, any indices defined within a **while** loop cannot contribute to independence, and are 'invalid' too.

To determine the degree of a process, the *Optimiser* first finds the subgraph of the 'hard' SDG, as above. Second, it finds the 'invalid' loops for each event, by comparing the frequencies of its vertices. Third, it strips out the calling frequency of each definition, by removing all loop variables preceding the first loop variable whose definition is allocated to the process. Finally, it determines the independence of each variable as described in Section 10.5.1 with an important modification: only 'valid indices' are accepted. Valid indices are valid loop variables, constants or parameters. Valid indices are those whose definitions have frequencies that do not include a **while** loop or an invalid **all** loop.

This approach deals correctly with the example of Example 7.5.2, which computes each employee's salary as a percentage of the total payroll. Its optimum process graph comprises two processes, the first containing the loop on 'Emp1', and second containing the loop on 'Emp2'. The values of 'Total' and 'Temp' are passed between them. It is important to keep these processes separate; the independent access algorithms do not allow two **all** loops in succession. The local array 'Temp' is important here. It is used to store a copy of 'Salary'. Without it, as in Example 7.5.1, both loops have to access 'Salary', and since 'Salary' is a state variable, they are allocated to the same process. Since 'Emp1' and 'Emp2' are invalid indices, no independence is possible.

However, the analysis of Example 7.5.3 is more problematic. Its purpose is to compute the average balance of all authorised customers. To do this, it computes the total balance in one loop, and counts the number of authorised customers in a second loop. As in the case of Example 7.5.2, the *Optimiser* allocates the two **all** loops to separate processes. The value of 'Total' is passed from the first process to the second. However, it would be theoretically possible to compute the values of 'Total' and 'Count' within the same loop, as in Example 7.5.4. This case may differentiated from the case of Example 7.5.2, which requires both loops. The opportunity for folding the loops can be detected by observing that no definition made in the second loop of Example 7.5.3 depends on any definition made in its first loop. However, if the *Optimiser* took this into account, and assigned both loops to the same process, the *Programmer* would need to know how to generate a process in which the two loop bodies were folded together. This would require more than the simple text-substitution mechanism associated with delayed procedure call. Consequently, the *Optimiser* currently keeps the loops separate.

10.5.3 Optimiser Heuristics

The two conditions under which it pays to merge processes 'X' and 'Y' are called 'subsumption', and 'promotion'.

Process X 'subsumes' process Y if X accesses every index that Y accesses. This means that if the attributes accessed by X and Y were stored in the same records, X would already need to access every record that Y would access. Merging X and Y is therefore bound to save accesses. This is an obvious saving if X must use random access, because the attributes used by Y are accessed at no additional cost. It is also a saving if independent access is used, because every record needs to contain a key, which is copy of its index value, so combining two records saves storing and accessing the second key. If the two processes are neighbours in the process graph, combining the processes also eliminates the cost of transmitting data between them.

X 'is promotable to' Y if the degree of parallelism of X either equals, or can be extended to equal, the degree of parallelism of Y. If X and Y have the same degree of parallelism, then the attributes they access can be accessed by the same process, and combined into the same table — provided that their indices are compatible. By eliminating one copy of each key, the total storage occupied is reduced, and this reduces the number of accesses needed. On the other hand, if the degree of X is not equal to the degree of Y, but extensible to it, it follows that X must access only local variables or parameters. In such a case, combining the processes avoids creating a trivial process for the local variables. There are several ways in which X could be involved:

In one case X could access a simple variable with one definition, as in:

```
X := W(i);
Y(k) := X;
```

X could be merged either with 'W(i)' or 'Y(k)', which are both assumed to be state variables here. However, once X has been merged with 'W(i)', the resulting process would have the same degree as 'W', and would no longer be extensible, so it is unlikely that it could be merged with the 'Y' process. Conversely, merging 'X' with 'Y(k)' would probably prevent it being merged later with 'W(i)'. In this instance, it does not matter which optimisation is chosen.

A second case is when 'X' is an accumulator variable, as in:

```
all i in index loop
   X := X+W(i);
end loop;
```

In this case 'X' should be merged with 'W' to avoid passing all the 'W(i)' values between processes.

The third case is when 'X' is broadcast, and inspected in a loop, as in:

```
all k in key loop
   Y(k) := Y(k)+X;
end loop;
```

Again, the motive for merging is to avoid having a trivial process for 'X'.

However, if there is a choice between merging 'X' with 'W(i)', as in the second case, or with 'Y(k)', as in the third case, the merge with 'W(i)' should have priority, because it reduces the number of procedure calls that must be passed between processes. These last two cases introduce an asymmetry into the merging of neighbouring processes. If process 'X' is promotable to 'W' or to 'Y', 'Y' depends on 'X', and 'X' depends on 'W', the optimiser will therefore consider merging 'X' with 'W' before considering merging it with 'Y'. This is in accordance with the principle of early evaluation discussed in Section 7.1, in which it was also noted that loop variables should be an exception. Because the *Parser* extends the names of loop variables with a distinctive prefix, they are easily recognised. The *Optimiser* never merges a simple loop variable with an upstream process. However, once the loop variable has been merged with other variables, that becomes a different matter.

As a result of these considerations, the optimiser merges processes 'X' and 'Y' according to the following priorities:

1. 'X' and 'Y' are neighbours, 'X' depends on 'Y', and either 'Y' subsumes 'X' or 'X' is extensible to 'Y'. (Except in the case that the only member of 'X' is a loop variable.)

2. 'X' and 'Y' are neighbours, 'X' depends on 'Y', and either 'X' subsumes 'Y' or 'Y' is extensible to 'X'.

3. There is no directed path between 'X' and 'Y', and either 'X' subsumes 'Y' or 'X' is extensible to 'Y'. (There is a symmetrical case, exchanging the roles of 'X' and 'Y'.)

A problem with these rules as they stand is that simple local variables are compatible with anything, including other simple local variables. There is a danger that the *Optimiser* will begin by placing all the local variables into a single process, then find that it is difficult to combine this process sensibly with any of the others. For example, if the *Optimiser* were to compose 'Stock' and 'Loans' given the example of Example 10.4.1, it would not be able to allocate them optimally, as in Example 10.5.1. The *Optimiser* therefore gives a very low priority to composing two processes whose degree is extensible. The practical effect is that local variables must be composed with state variables before they can be composed with other local variables. As a result, such compositions could never be made except in a trivial event specification that made no reference to any state variable.

Another problem in optimisation is that the graphs constructed by the *Analyser* are usually much more complex that those that have appeared in diagrams in this text. The diagrams typically suppressed local variables and parameters and showed only the relationships between state variables. Those constructed by the *Analyser* not only include local variables, but variables representing array elements. What is more, there may be several definitions of each variable. As a result, two state variables that are linked by a single edge in a diagram in the text may be linked by a long compound path in the *Analyser* graph. For one optimisation step on a diagram, the *Optimiser* may make several steps. Since most optimisation steps involve merging neighbouring vertices, the *Optimiser* has no long-range goal and succeeds only locally. Indeed, this is the main characteristic of a greedy algorithm. It therefore seems a matter of chance whether the *Optimiser* will succeed globally. To help counteract this, the *Optimiser* always chooses to merge two non-extensible processes when it can.

As a result of these considerations, the *Optimiser* gives highest priority to merging non-extensible processes, second highest priority to merging an extensible and a non-extensible process, and lowest priority to merging two extensible processes. Within each of these three groups, it then applies the 3 rules given earlier. As a result, it has 9 priorities altogether.

10.6 The Generator

The *Generator* reads the first part of the *Optimiser's* output, consisting of the topological ordering assigned to the processes. This is shown in Example 10.6.1. This ordering associates each lexical definition with a process. The *Generator* also reads the abstract syntax tree created by the *Parser*. The output of the *Generator* is a labelled syntax tree, as shown in Example 10.6.2.

```
[([user],[lex([global/library/d,*],1),
   lex([global/library/d,input/borrow/u],2),
   lex([global/library/d,input/borrow/u],3),
   lex([global/library/d,input/reshelve/u],4),
   lex([global/library/d,input/reshelve/u],5),
   lex([global/library/d,loop/audit/u],6),
   lex([input/borrow/u],1),
   lex([input/reshelve/u],1),
   lex([local/audit/loans],0),
   lex([local/audit/loans],1),
   lex([local/audit/loans],2),
   lex([loop/audit/u],1),
   lex([output/report/audit],1)]),
 ([title],[lex([global/library/c,*],1),
   lex([global/library/c,input/borrow/t],0),
   lex([global/library/c,input/borrow/t],2),
   lex([global/library/c,input/borrow/t],3),
   lex([global/library/c,input/buy/t],7),
   lex([global/library/c,input/buy/t],8),
   lex([global/library/c,input/reshelve/t],5),
   lex([global/library/c,input/reshelve/t],6),
   lex([global/library/c,loop/audit/t],9),
   lex([input/borrow/t],1),
   lex([input/buy/t],1),
   lex([input/reshelve/t],1),
   lex([internal/borrow/expn_1],1),
   lex([local/audit/stock],0),
   lex([local/audit/stock],1),
   lex([local/audit/stock],2),
   lex([loop/audit/t],1)])].
```

EXAMPLE 10.6.1: PART OF THE OUTPUT FROM THE OPTIMISER

The *Generator* places the topological number and degree of independence of each process alongside each definition in the syntax tree, and reconstructs the text of the specification. Because the specific forms of expressions are unimportant to dependence analysis, the *Parser* does not preserve them, so the reconstructed specification represents each expression as a call on an unknown function, denoted by 'fn_1', 'fn_2', and so on. However, it is clear that there is nothing in principle to prevent the reconstruction being faithful to the original. As explained in Chapter 7, there would then be sufficient information in Example 10.6.2 to generate detailed process specifications.

```
o. ('Process_1',[title])
^o ('Process_2',[user])

package body library is
 d:array (user) of natural;                        -- Process_2, [user]
 c:array (title) of natural;                       -- Process_1, [title]
 procedure borrow(u:user;                          -- Process_2, [user]
                  t:title) is                      -- Process_1, [title]
 begin
  if fn_1(c(t)) then                               -- Process_1, [title]
   c(t) := fn_2(c(t));                             -- Process_1, [title]
   d(u) := fn_3(d(u));                             -- Process_2, [user]
  end if;
 end borrow;
 procedure reshelve(u:user;                        -- Process_2, [user]
                    t:title) is                    -- Process_1, [title]
 begin
  c(t) := fn_4(c(t));                              -- Process_1, [title]
  d(u) := fn_5(d(u));                              -- Process_2, [user]
 end reshelve;
 procedure buy(t:title) is                         -- Process_1, [title]
 begin
  c(t) := fn_6(c(t));                              -- Process_1, [title]
 end buy;
 procedure audit is
  loans:natural;                                   -- Process_2, [user]
  stock:natural;                                   -- Process_1, [title]
 begin
  all t in title loop                              -- Process_1, [title]
   stock := fn_7(stock,c(t));                      -- Process_1, [title]
  end loop;
  all u in user loop                               -- Process_2, [user]
   loans := fn_8(loans,d(u));                      -- Process_2, [user]
  end loop;
  report.audit(stock,loans));                      -- Process_2, [user]
 end audit;
end library;
```

EXAMPLE 10.6.2: THE OUTPUT OF THE GENERATOR

10.7 Conclusions

The *Designer* tool is successful in so far as it achieves correct results. Apart from its obvious deficiencies in generating process specifications — which could be remedied, given sufficient programming resources — there are two other problems. The first is that it is slow. It takes the *Designer* 20 seconds (on a Macintosh LC630) to transform Example 10.2.1 into the output of

Example 10.6.2. Worse, it takes it 27 times longer (9 minutes) to analyse a specification that is about 3.5 times longer. This suggests that some parts of the *Designer* have complexity $\mathbf{O}(n^3)$, where n is the number of words in the specification. Since the number of vertices in the SDG produced by the *Analyser* is roughly proportional to the length of the specification, except in those cases where the specification contains loops, the cause of the problem would seem to be the *Optimiser*. This hypothesis is supported by timing the various phases of the *Designer* independently.

The reason is that the number of optimisation steps is roughly proportional to the number of vertices in the CPG, and each optimisation step involves a pair of vertices. Since sets of vertices or processes are stored as lists, there are many situations in which the *Optimiser* must scan lists. For example, an optimisation step involves scanning a list of edges, and for each edge, the properties of the processes it connects must be retrieved from other lists. What is more, after each successful optimisation, the *Optimiser* must find the transitive root of the resulting graph. As explained earlier, this requires edges to be eliminated that correspond to composite paths. The current implementation makes a depth-first search to test each edge. As Figure 10.7.1 and Figure 10.7.2 show, an optimisation step may introduce a path that makes it necessary to delete an edge that is remote from the site of the optimisation. In Figure 10.7.1 the edge from {A} to {E} is part of the transitive root, because there is no other path from {A} to {E}. But after {D} and {F} are composed, as in Figure 10.7.2, there is a compound path from {A} to {E} via {D,F}, so the edge from {A} to {E} becomes redundant.

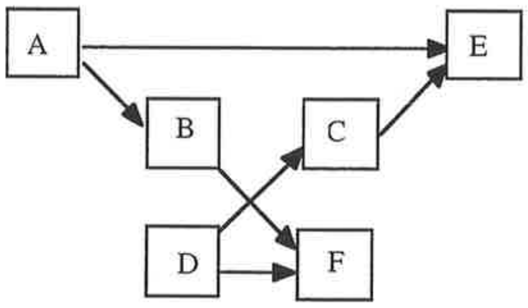

FIGURE 10.7.1: BEFORE OPTIMISATION

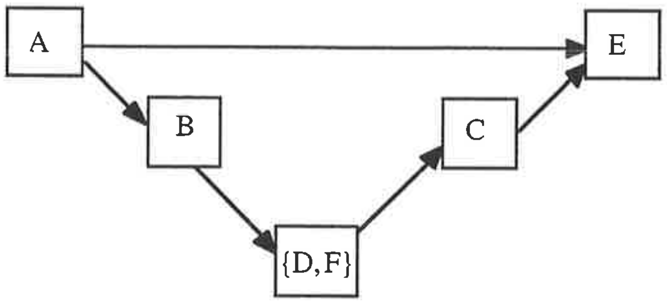

FIGURE 10.7.2: AFTER OPTIMISATION

Given these difficulties, it is perhaps surprising that the time taken to optimise a process graph of n vertices does not grow faster than $\mathbf{O}(n^3)$. The saving grace here is that the vertices of the transitive root of a graph tend to have low degree, usually having no more than two incident edges. Thus, although the number of edges of a graph is bounded by the square of the number of its vertices, in practice, the number of edges is roughly equal to the number of vertices. Therefore, searching for a composite path takes time roughly proportional to the number of vertices. Since this must be done for each edge, and there are roughly as many edges as vertices, finding the transitive root takes time proportional to the square of the number of vertices. Since a transitive root must be found after each optimisation step, the complexity of the *Optimiser* is roughly $\mathbf{O}(n^3)$.

There is little doubt that the *Optimiser* could be made faster by using more sophisticated data structures. However, more efficient structures would be hard to implement in Prolog, and are a subject for future work, and not within the scope of a prototype. An interesting possibility would be to find the transitive root by a novel adaptation of Eve and Kurki-Suonio's transitive closure algorithm, whose execution on an acyclic graph evaluates the transitive closure of each vertex on its postorder visit. The algorithm could be modified to find only the indirect successors of the vertex. It would then be possible to inspect its direct successors and delete those edges that were redundant. The efficiency of this algorithm would be close to that of Eve and Kurki-Suonio's transitive closure algorithm.

```
with report;
generic
   type product is private;
   type money is range <>;
package personnel is
   procedure top_dog (who : employee);
end personnel;
package body personnel is
   Boss : array (employee) of employee := (others => null);
   Salary : array (employee) of money := (others => 0);
   procedure top_dog (Who : employee) is
   begin
      while Boss (Who) /= null loop
         Who := Boss (Who);
      end loop;
      report.put (Salary (Who));
   end top_dog;
end personnel;
```

EXAMPLE 10.7.1: A SPECIFICATION WITH A WHILE LOOP

In only one instance did the *Designer* produce a result that was sub-optimal. This was in its analysis of a specification incorporating a **while** loop, based on Example 7.3.2, and shown in Example 10.7.1. It assigns 'Boss' and 'Salary' to separate processes. The 'Boss' process uses random access, but the 'Salary' process is independent with respect to 'employee'. In fact, it is probably better to combine the two processes; the index of 'Salary' is the same as the index of 'Boss' on the last iteration of the loop. Thus, by packing 'Boss' and 'Salary' into one record, the access to 'Salary' could be free. Although this would uselessly retrieve 'Salary' on each iteration of the loop, in practice there would still only need to be one disk access per iteration. The reason why the *Designer* cannot find the better solution is because the *Analyser* uses the single definition method. The definition of 'Who' following the loop is a *new* definition produced by merging that entering the loop, and the two dynamic definitions created within it. It is therefore equal to none of them, and the ‘Boss’ process cannot subsume the ‘Salary’ process.

11. Discussion

This chapter discusses the implications of the thesis, and its relationship to other work.

11.1 Comparison with Other Methodologies

There are many methodologies that have been developed for information system design. [Olle *et al.* 1991] list 32 methodologies that are well documented, many of which are available commercially, either as CASE products or through special tuition. The methodologies differ in which analysis products they consider important, and in the conventions (diagrams, charts) used to represent them. They also differ in their perspective: data-oriented, process-oriented or behaviour-oriented. In principle, an almost unlimited number of methodologies could be devised.

In the face of this complexity, the following subsections deal with only three methodologies, which seem to be those most strongly related to the Canonical Decomposition Method presented here. They are among the most widely used: Entity-Relationship (E-R) Modelling, Structured Analysis and System Specification (SASS), and Jackson System Development (JSD). ([Floyd 1986] compares them.) They are examples of data-oriented, process-oriented and behaviour-oriented perspectives.

11.1.1 Entity-Relationship Modelling

One of the features of the specification language presented in Chapter 2 is that the database is modelled by FD's, whose syntax is similar to that of arrays. We say that 'Y' is functionally dependent on 'X', if, given 'X', it is possible to deduce the unique associated value of 'Y'. It is a prerequisite of the specification process given here that the FD's have already been recognised by the specifier.

One way of discovering FD's is through Entity-Relationship Modelling [Chen 1976]. This data-oriented technique requires that, in consultation with the client, the database designer should first identify the kinds of entities the system should model. Each entity is modelled by a table within the database. Second, the designer should identify the attributes associated with each kind of entity, which become the columns of the tables. Third, the designer must discover which sets of attributes uniquely identify an individual entity. These sets are called candidate keys. One candidate key must be chosen to be the primary key of the table. Fourth, the designer should consider the relationships between the entities that have been identified. Figure 11.1.1 shows a simple Entity-Relationship (E-R) Diagram for an order-entry system such as that discussed in earlier chapters.

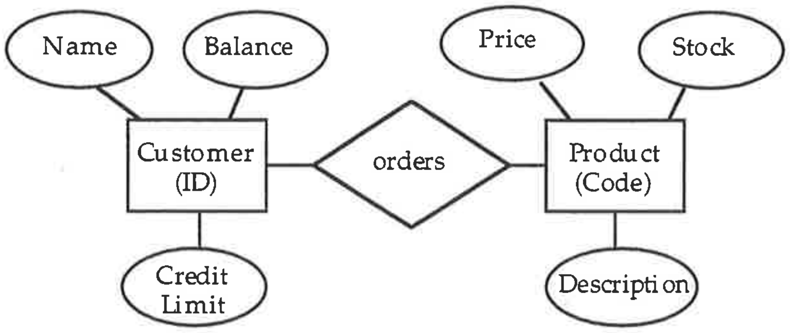

FIGURE 11.1.1: AN E-R DIAGRAM FOR ORDER PROCESSING

In Figure 11.1.1, the designer has discovered that 'Customer' entities are identified by 'IDs' and have 'Names', 'Credit Limits' and 'Balances' as attributes. Similarly, 'Product' entities are identified by 'Codes', and have 'Prices', 'Stocks' and 'Descriptions' as attributes. From this, we may immediately conclude that 'Name', 'Balance' and 'Credit Limit' are functionally dependent on 'ID', and that 'Price', 'Stock' and 'Description' are functionally dependent on 'Code'. It is easy to model these properties in a database. For example, each product may be represented by a row of a 'Product' table whose primary key is 'Code', and whose non-key attributes are 'Price', 'Stock' and 'Description'.

The designer has also recognised that there a relationship between the 'Customer' and 'Product' entities, of the form 'Customer *orders* Product'. It is then necessary to classify the 'orders' relationship as one to one, many to one, one to many, or many to many. A *one-to-one* relationship would mean that a given customer is always associated with the same product. If that were the case, it might indicate that either some customers were products, or some products were customers, or customers and products were the same thing. A *many-to-one* relationship would mean that each customer would be able to order at most one product. Conversely, a *one-to-many* relationship would mean that each product could be ordered by at most one customer. None of these situations are likely to arise in practice. The most likely situation is a *many-to-many* relation, where each customer could order many products, and each product could be ordered by many customers.

The type of the relationship determines what FD's are present. A one-to-one relationship implies both that 'Code' is functionally dependent on 'ID' and that 'ID' is functionally dependent on 'Code' — perhaps through the identity function. A many-to-one relationship implies only that 'Code' is functionally dependent on 'ID'. A one-to-many relationship implies that 'ID' is functionally dependent on 'Code'.

The case of a many-to-many relationship is more interesting. 'Code' is not functionally dependent on 'ID', nor is 'ID' functionally dependent on 'Code'. To represent a many-to-many relationship in a database, it is necessary to introduce a new 'Orders' table having 'Code' and 'ID' as attributes. Thus, the fact that a customer orders several products would be represented by the 'Orders' table containing several rows having the same customer ID but different product codes. Conversely, the fact that a given product is ordered by several customers is modelled by the 'Orders' table containing several rows with the same product code but different customer

ID's. Because the 'Code' and 'ID' attributes of the 'Orders' table refer to primary keys in the 'Product' and 'Customer' tables, they are called 'foreign keys'.

Once the designer recognises a many-to-many relationship, it often becomes apparent that the database table that represents it should contain additional attributes. In this case, each 'Orders' row should contain the quantity of the product that the customer orders. It also turns out that no set of attributes in the 'Orders' table is sufficient to identify an order uniquely; the same customer might order the same quantity of the same product on several different occasions. In such a situation it is necessary to use some kind of serial number or time-stamp to identify each order uniquely. In this way, the many-to-many relationship becomes an entity in its own right.

A weakness of Entity-Relationship Modelling is that it has a rather static view of the world: essentially a snapshot. It does not distinguish clearly between entities such as products and customers, which are likely to be represented within the database, and entities such as orders, which are events that update the database. Similarly, the SQL language [Date 1993], which is the standard language for describing and manipulating databases, has no means for treating a row of a table as an event, except in very simple cases. Whenever a database table has no natural primary key, and rows have to be identified by serial numbers or time stamps, this is usually a sign that the rows concerned represent events. Therefore Entity-Relationship Modelling cannot be regarded as a complete methodology, but it is an important part of several other methodologies, e.g., [Weinberg 1980].

Although it is not strictly part of Entity-Relationship Modelling as such, modelling is usually followed by 'database normalisation'. By this is meant that an attempt is made to design the database in such a way as minimise redundant information. The idea is that it should be impossible to introduce anomalies when the database is updated. For example, if the name of each customer were stored twice, it would be possible for the two copies of the name to differ, and in any case, it would be more complicated to update two copies rather than only one. For most problems, this goal is easily realised, except for foreign keys: attributes that link to the primary keys of other tables. For example, since each 'Orders' row refers to a customer row by means of its ID, any update that changes the ID of a customer row should also change all the corresponding ID's in the 'Orders' table. However, most modern database management systems can propagate such changes automatically, provided that the foreign keys are declared as part of the database schema.

A potential drawback of normalisation is that it requires all attributes with the same primary key to be placed in the same table, because this saves storing redundant copies of the primary key. The consequence is that it tends to suggest system designs in which all the attributes functionally related to a given primary key are updated within the same process, whereas previous chapters have presented cases where the attributes are best split between processes, e.g., in the design of Figure 6.6.4 (Page 146). To be fair, although normalisation requires all the attributes of a given key to share the same table, it does not forbid the same table to be updated more than once. Thus, even if there is only one customer file and one product file, Figure 6.6.4 is still applicable. It simply means that each update uses certain attributes and ignores the others. However, splitting attributes between files is more efficient in this example, because fewer data blocks need to be read or written.

In summary, we may regard Entity-Relationship Modelling as complementary to CDM; it provides a means for identifying the state variables the system will use. However, Entity-Relationship Modelling does not guard against the problem observed in Section 6.6, that given a certain requirement, the choice of system variables can critically affect the design and efficiency of a system.

11.1.2 Structured Analysis and System Specification

How does CDM compare with probably the most widely used method of designing information systems, namely, Structured Analysis and System Specification (SASS) [DeMarco 1978]? The basis of SASS is the data-flow diagram (DFD). SASS is one of a family of 'Structured Design' methods promoted by the Yourdon school [Yourdon & Constantine 1978, Weinberg 1980, Chapin 1981, Colter 1982, Connor 1981, Delisle *et al.* 1982, Richter 1986]. Collectively, this school has proved adaptable and eclectic, for example, adopting Entity-Relationship Modelling and database normalisation into its armoury, so that it is difficult to specify exactly what Structured Design *is*. SASS is perhaps the most purely process-oriented Structured Design method.

DFD's roughly correspond to what this thesis calls process graphs. The primary difference is that process graphs allocate state variables (database attributes) to processes, whereas DFD's show data stores and processes as separate objects. Figure 11.1.2 shows a process graph, and Figure 11.1.3 shows its corresponding DFD. In Figure 11.1.3, it is assumed that the database has been normalised, so that there is a table for 'Product' information, a table for 'Customer' information, and a table for 'Back Order' information. The DFD shows that an external source, 'Customers', gives rise to a stream of 'Orders' that flows into a process called 'Check Offered and Price'. This process reads the 'Stock Records' table but does not update it. It also sends a data flow to a process called 'Check Authorisation and Credit' that reads 'Customer Accounts'. 'Check Authorisation and Credit' then sends a data flow to 'Check and Update Stock' which updates 'Stock Records'. 'Check and Update Stock' sends data flows to 'Update Balance', which updates 'Customer Accounts', and 'Record Back Order' which updates 'Back Orders'. (Strictly speaking, all the data flows in the diagram should have been given meaningful labels, such as 'Priced Orders', etc.)

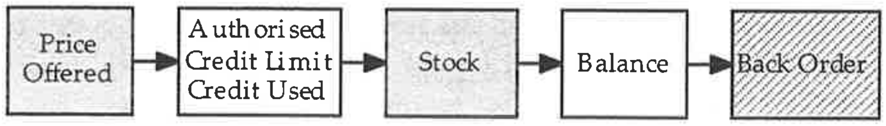

FIGURE 11.1.2: PROCESS GRAPH FOR ORDERS WITH CREDIT USED

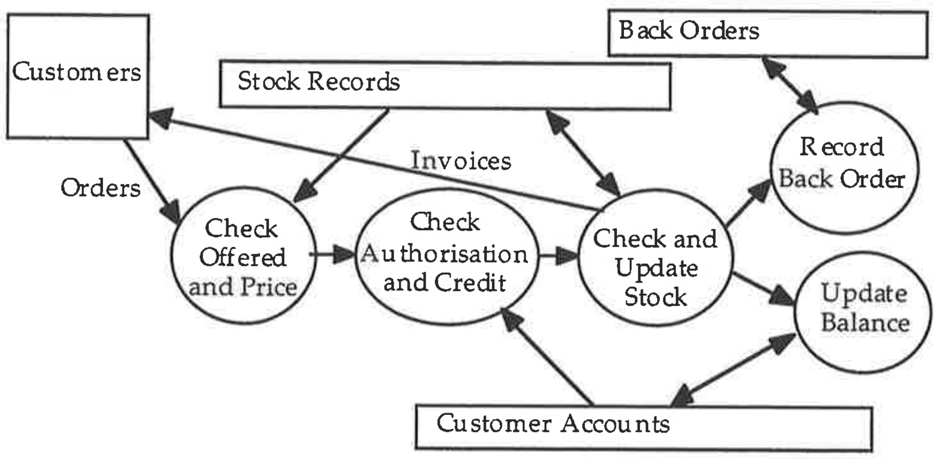

FIGURE 11.1.3: DATA-FLOW DIAGRAM FOR ORDERS WITH CREDIT USED

It is clear that all the process graphs in this thesis could be derived by using SASS. Unfortunately, so could many others, most of them wrong. The DFD of Figure 11.1.3 demonstrates some of the difficulties of the SASS method.

Again, database normalisation suggests that all attributes sharing a given key should be placed in the same table, so that the DFD does not distinguish, for example, between the 'Stock Records' attributes accessed by 'Check Offered and Price' and those updated by 'Check and Update Stock'. It is therefore not clear that the DFD represents a valid batch processing system. If an attribute updated by 'Check and Update Stock' were also inspected by 'Check Offered and Price' only its initial state would be observed. In fairness, it would be possible for the designer to split 'Stock Records' into 'Price Information' and 'Stock Information', but although SASS allows this, it conflicts with SASS's emphasis on database normalisation.

A second difficulty is that the DFD shown only shows the processing for 'Orders'. Once other event types are considered, there is difficulty in drawing the DFD. If a new set of processes is drawn for each type of event, the DFD does not show how the sets of processes should be integrated to implement the system as a whole. To be implemented as a batch system, the processes for each type of event would have to be merged. But once they are merged, the processes gain complex descriptions, since the processing for each kind of event is different. This is in conflict with a basic rule of SASS that requires each process to have a simple, meaningful description. The only simple description that can be given is the rather vague "Inspect and update 'Offered' and 'Price', etc.", which is effectively how a CDM process graph is labelled. A similar labelling problem affects the data-flows, which must now carry messages of many kinds. These problems are not so much a fault of SASS itself as due to the nature of batch processing. As we have seen in earlier chapters, the event procedures within batch processes are made up of almost meaningless fragments of event specifications, and the data-flows are merely calls on these procedures. Nevertheless, to design a batch system, the processes for each kind of event *must* be merged, an action that SASS calls 'regrouping simple processes' — although SASS gives no specific guidance about how this ought to be done.

Although the DFD resulting from analysing the problem may contain cycles, the DFD of a batch implementation must be acyclic. It is clear that if a designer is to use SASS successfully to design a batch system, there must be some stage at which the cycles in the DFD are given special treatment. Unfortunately, there seems to be no formal design stage at which this occurs. In text books describing SASS [DeMarco 1978, Gane & Sarson 1979], the essential step seems to occur when the system DFD is subdivided into programs, but the method is never explained, or even mentioned. In some texts, it is simply stated that cycles directly between processes are forbidden [Hawryszkiewycz 1994]. It is also unclear when data-flows can be thought of as queues. In many contexts it is obvious that they can be implemented as transfer files, as inter-office memoranda, or other forms of messages; in others it is not. The designer is supposed to decide somehow when queues are valid.

A typical program within a system designed using SASS has a number of ‘afferent branches’ that merge data from various sources into a nexus called the ‘central transform’. From the central transform, data spreads out to its various destinations along ‘efferent branches’. It is the function of the afferent branches to transform data from its stored representation into an abstract form most suitable for use by the central transform, and also to collate corresponding items from different sources. The efferent branches arrange the results generated by the central transform into formats suitable for display or storage. The central transform itself changes abstract inputs into abstract outputs. It is quite common for a central transform to be ‘transaction directed’, that is, to consist of a **case** statement that directs each type of transaction to a separate procedure for processing.

From this description, it will be seen that a typical program is assumed to have an acyclic flow of data, inward from the afferent branches to the central transform, then outward along the efferent branches. It is also the case that a program must correspond to a subgraph of the DFD. For example, in the DFD of Figure 11.1.3, it would be possible for ‘Check Offered and Price’ to constitute a program. Its afferent branches would flow from ‘Orders’ and ‘Stock Records’ and its efferent branch would flow into a transfer file called ‘Priced Orders’. Since it is always possible to segment a graph into acyclic subgraphs, SASS must always succeed. The problem is that instead of SASS insisting that cycles must be placed within a single process, as this thesis proves that they should be, it actually tends to spread them across several processes.

In defence of SASS, it does recognise the paramount importance of data flow in system design. However, it is not clear from the literature on SASS whether data flows should be derived from some underlying notion of dependence. If so, the idea has not been formalised. In short, it must be said that although SASS *allows* correct design, it also allows or even encourages wrong design. Its successful use relies heavily on the designer’s experience.

An interesting variant of SASS for real-time systems design is described by [Ward & Mellor 1978]. Primarily, it adds to SASS by introducing notations for control aspects of a system. Essentially, data flows are controlled by the actions of finite state machines. There is a clear distinction between control signals, which carry no data, and data-flows, which carry no control information. This makes it impossible to describe delayed or remote procedure calls, which carry both control and data. Also, rather curiously, the method allows control signals to be placed in queues, whereas data-flows may not. It makes it possible to implement semaphores [Dijkstra 1968], but makes it almost impossible to describe a batch system, or any other set of separable processes linked by queues. This might be defended if such queues were unusual features of real time systems. In fact they are commonly used, as evidenced by Monitors [Hoare 1974], and the Mascot methodology [Simpson 1986].

11.1.3 Jackson System Development

Like SASS, Jackson System Development (JSD) concentrates on the data flows between processes, which it calls 'datastreams'. Datastreams have the same properties as queues. However, JSD is more formal than SASS, and has more in common with CDM. Like CDM, JSD recognises that the purpose of a database is to record the states of objects, which it calls 'entities'. Likewise, the purpose of datastreams is to pass messages between entities to update or inspect their states.

[Olle *et al.* 1991] classify JSD as process-oriented, but this seems to be taking a rather superficial view of it. JSD is really a behaviour-oriented method, but it describes behaviours by processes that generate them [Kobol 1987]. These so-called 'model processes' describe the behaviour of real-world entities. They are only indirectly related to business or computer processes.

Some aspects of JSD are a useful complement to those described here. JSD views entities as long-lived, communicating processes. From the point of view of the business, each customer is a process that emits orders, payments and so on. Likewise, each product *item* is a process with a history of being purchased, ordered, sold, delivered, and so on. The simplest implementation of a system would be to model each real-world process directly by a corresponding model process. However, mapping these processes onto active tasks would overtax the largest computer; there are simply too many of them. Even given sufficient resources, direct modelling would still be inefficient; processes that represent customers or product items, for example, are active for only a tiny fraction of the time. In addition, their activation records must be persistent.

In the absence of a technology that implements a large number of low-activity, persistent processes efficiently, it becomes the designer's task to devise a good way to store their activation records in a database. Thus, the activation record of a customer model process, say, is represented by a row in a customer table. An order event may be interpreted as modelling the interaction of the customer process with a product process, which is modelled by another row in the database. Events may be seen as the means by which model processes and the real-world entities they model are brought into step. JSD aims to transform the set of model processes into a set of computer processes. Describing the behaviour of real-world entities by model processes is a means of mapping all system design elements to processes. Thus, system design becomes a series of process to process transformations.

A strength of JSD is its emphasis on event sequence. As already mentioned, each entity is seen as a process. The events a process can interact with must occur in certain specific sequences: for example, a product cannot be sold until after it has been manufactured. The valid sequences may be modelled by finite-state automata, whose states are recorded in the database. Such states have names such as 'in production', 'in stock', etc. (Typically, the state diagram of the automaton is strongly-connected, so that the associated entity can be forced into any desired state — perhaps to correct an update carried out in error. This requirement often prompts the discovery of event types that had been overlooked in earlier analysis.)

An event typically interacts with more than one entity. For the event's procedure to be applicable, all its interacting entities must be in states where the event is possible. An important aspect of JSD is that this is the *only* constraint on event sequence. If the existing set of entities does not sufficiently constrain the sequence of events, a new entity must be introduced to constrain the sequence further.

As an example, consider a student enrolment system. Once students have been admitted to a degree, it is possible for them to enrol in subjects or withdraw from them as often as they wish. Likewise, subjects may be enrolled in or withdrawn from by students as needed. Each enrolment or withdrawal involves an interaction between a specific student and a specific subject. However, the automata of the student and subject entities alone are unable to express the constraint that a student may not withdraw from a course without having previously enrolled in it. Even using push-down automata to count the numbers of enrolments made by students does not solve the problem. It would still allow a student to enrol in one subject, but withdraw from a different one. To enforce the constraint properly, it is necessary to introduce enrolment entities, corresponding to enrolments of specific students in specific subjects. An enrolment entity needs to have at least two states: active and inactive. In the inactive state, only enrolment events are allowed; in the active state, only withdrawals are allowed.

(There is a useful parallel here between JSD and Entity-Relationship Modelling. The relationship between students and subjects is many-to-many. In different ways, both methods recognise the need to model the '(student, subject)' pairs present in the relation.)

Of course, sequences of interactions between processes can also be modelled by Petri Nets [Maiocchi 1985, Tse & Pong 1986, Tse & Pong 1989]. It is easy to transform a set of interacting processes into a Petri Net model. However, it is not so easy to decompose a Petri Net model into a set of interacting processes. In this respect, the JSD model captures aspects of the real-world that the Petri Net model doesn't.

Once the analyst has recognised the sets of entities and events needed, the next step in JSD is to draw a System Specification Diagram. This is a form of DFD in which pathways are provided to pass messages to and between entities. Later, the System Specification Diagram is transformed into an implementation scheme, in the form of a Network Diagram. The various design stages use graphical conventions to express the designer's intentions, but offer little direct help to the designer.

Although JSD predates the wave of interest in object-oriented programming, it is essentially an object-oriented technique [Masiero & Germano 1988]; the entity processes are 'objects', and the events are 'methods'. As a result, some comments made below about JSD can apply equally to other object-oriented design methods [Booch 1991].

In the light of the Canonical Decomposition Method, there are two shortcomings of the JSD approach. The first is that the object-oriented viewpoint encourages the idea that all the attributes of a given object should be accessed by a single process that models the entity. Thus, the order processing example should model two sets of processes, one for customers, and the other for products. The result of modelling the order processing example is shown in the System Specification diagram of Figure 11.1.4.

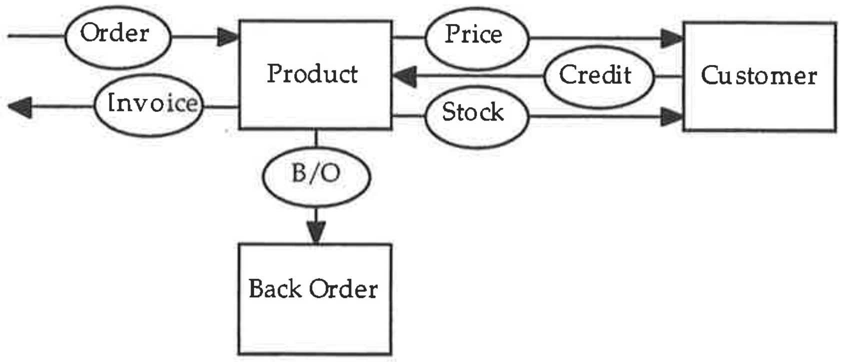

FIGURE 11.1.4: SYSTEM SPECIFICATION DIAGRAM FOR ORDERS WITH CREDIT USED

Figure 11.1.4 shows that an 'Order' message enters the system, and is passed to a 'Product' process, which returns an 'Invoice' message. To do this, it carries out a dialogue with the 'Customer' process, involving 'Price', 'Stock' and 'Credit' messages. It may also send a 'B/O' message to a 'Back Order' process. The SSD does not specify the order in which the messages occur.

(There is a subtle difference between the ways customers and products are modelled. Each customer row models *one* customer, whereas each product row models a *set* of indistinguishable products of the same kind. Therefore, a customer row records the state of a single customer, but a product row records the states of many products. This is done by recording the *numbers* of products — of the same kind — that are in the 'in stock' state, 'on order' state, 'sold' state, and so on. This leads naturally to 'conservation laws', the idea that the total numbers of items should be conserved as they move from state to state.)

Using JSD, a designer might then elaborate the System Specification Diagram into a Network Diagram that resembles Figure 11.1.2, perhaps that shown in Figure 11.1.5. However, there is no guarantee that the designer will consider this option, which is actually counter-intuitive because it requires that what was originally seen as a 'Product' entity should become both a 'Product' and 'Inventory' entity, and what was originally a 'Customer' entity should become both a 'Customer' and 'Account' entity. Such distinctions might be justified by stating that, 'A Product record contains generic information about a *kind* of item that is sold, whereas Inventory refers to *individual* items.' Or, 'Although in this system each Customer has exactly one Account, this need not be true in general, so we must be careful to distinguish Accounts from Customers.'

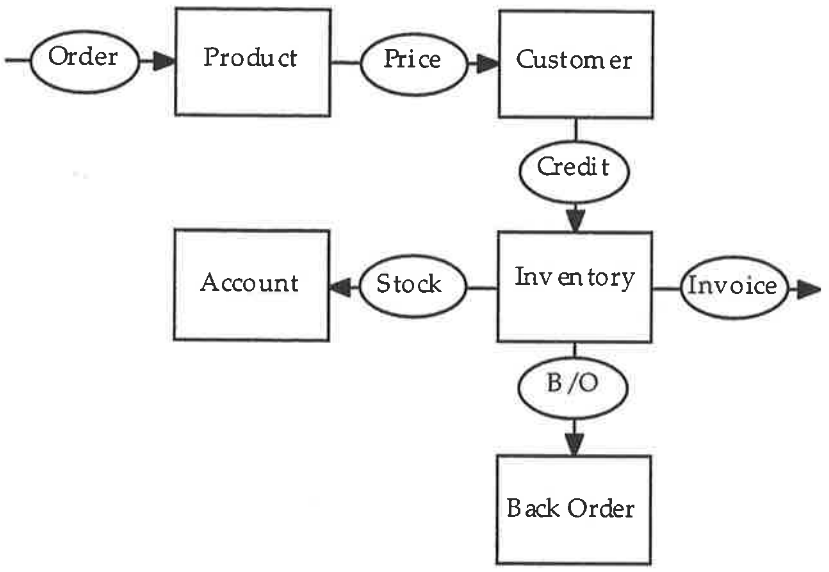

FIGURE 11.1.5: NETWORK DIAGRAM FOR ORDERS WITH CREDIT USED

In short, JSD has the same difficulty with cyclic flows that SASS has, and like SASS, it has no formal means of dealing with them. Perhaps JSD would be improved by a rule that suggested that the designer should try to eliminate cycles by splitting entities where necessary.

Another dubious aspect of JSD is its use of 'state-vector connections', by which one model process may directly observe the state of another, without the use of a datastream connection and its attendant delay. Considering the implementation methods described in Chapter 3, and the discussion of real-time equivalence in Chapter 4, a state-vector connection cannot be regarded as valid unless there is no intention of preserving real-time equivalence. Typically, a state-vector connection is used by a reporting process — in which case the process is likely to report an inconsistent state of the database. Such reports are best regarded as providing meta-level information about the implementation itself, rather than the system that it models. Unfortunately, examples of the use of JSD seem to use state-vector connections much more freely than they should [Sutcliffe 1988].

On the positive side, JSD does recognise some aspects of system design that CDM considers important. One of these is the need sometimes to split a system into modes, which can eliminate cycles by partitioning the kinds of events into subsets. JSD calls this 'process dismemberment', in the sense that a process (object) that is being modelled, which can accept a stream of events (methods) of all kinds, is implemented by two or more model processes that can only accept particular subsets of events.

Another strong connection between CDM and JSD is its associated program design technique: Jackson Structured Programming (JSP) [Jackson 1975, Storer 1987]. JSP is well suited to the design of programs whose structure is dominated by the data structures of their inputs and outputs, and is therefore applicable to the design of update algorithms.

In JSP, a program's structure is determined by the data structures it manipulates. Typically, a program works its way systematically through some input data structures while systematically constructing some output structures. Some sets of structures lead to satisfactory program designs, but other sets have no direct solution. Where the order of data elements in one structure differs profoundly from the order of corresponding elements in another there is no direct way of deriving a program structure. These situations are called 'ordering clashes'. There are two ways to resolve ordering clashes: by providing random access to one or more structures, or by sorting one or more of the structures so that their corresponding elements are in the same sequence. These are exactly the same means used within CDM to deal with process incompatibilities. First, although CDM attempts to use independent-access processes whenever possible, if a process is allocated some incompatible attributes, it must resort to random access. Second, in order to prevent incompatible attributes being allocated to a process, CDM prefers to use two processes linked by a data flow. Typically, the data flow is sorted on input to the downstream process to allow it to use independent access. Thus, there is a close parallel between the JSP's notion of 'ordering clash' and CDM's notion of 'incompatibility'.

11.1.4 Advantages of the Canonical Decomposition Method

How do SASS and JSD rate in the light of CDM? Essentially, their treatments of cyclic flows are too informal for them to be used to derive valid batch-processing systems. It might be argued that this is because of their flexibility, in that the methodologies are not restricted to one class of problem. There are two things wrong with this argument: First, the methods are capable of producing incorrect results. Second, formalising a design process does not necessarily reduce its flexibility. CDM produces a canonical process graph, from which all possible valid designs can be derived by merging its minimal processes. It distinguishes between those situations where processes are strongly-connected, separable, or independent. It therefore clearly distinguishes when processes have to be closely coupled, may be linked by queues, or can operate in parallel without contention. The only way in which CDM lacks flexibility is that this thesis has concentrated on one particular cost function for optimisation: one that maximises independent access. In principle, any other design objective could be expressed by some suitable cost function. In practice however, it might be harder for the designer to specify the cost function than to choose the optimal processes by intuition.

CDM, like JSD, is superficially process-oriented, in that its specifications consist mainly of the bodies of event procedures. However, also like JSD, it is really a behaviour-oriented technique. Event procedures have little to do with any business or computer procedures that are likely to be observed except in a small business. They describe details of what may be parts of several business processes. The business and computer procedures that CDM uses are chosen from a restricted set of paradigms: random access, sequential access and parallel access updates. The purpose of CDM is to choose a suitable process network, then tailor these standard procedures to fit.

This is a respect where it differs strongly from JSD and SASS. Although both these methods also operate by transforming procedures, the transformations they make are hard to do. For example, JSD requires the designer to constantly switch between a process viewpoint and a state-transition viewpoint. In contrast, delayed procedure call relies only on simple text substitution. This simplicity results from taking a state-oriented view of behaviour above the event level, but a procedural view below it. The state of the database constrains event sequence, whereas events are modelled by procedures.

CDM is not a complete methodology. It does not say how the FD's and event specifications of a system specification are derived. Chapter 6 made it clear that this is an important task that can affect the efficiency of the design, often involving negotiation with the client. The author's opinion is that these early stages of analysis are well served by adapting some techniques from JSD. The most useful element is to model entities by finite-state automata. This leads to the discovery of the kinds of events that are needed and suggests conservation laws that need to be obeyed.

11.2 The Specification Problem

We have seen that a canonical form of system design can be derived rigorously from a specification. We have also seen that minor changes to a specification can cause major changes to the resulting design. Some of these changes make no difference to the observable behaviour of the system. For example, Section 6.6 showed how a system may be implemented more efficiently by changing the way its state is stored. Is it possible that there is some form of specification that eliminates this problem?

11.2.1 Data Flow as a Starting Point

Whereas CDM derives data flows from the dependences implied by an algorithm, it seems to be an implicit assumption of the SASS and JSD methodologies that data flows are somehow the givens of the problem — or even that they are something the designer can invent. Is there some way in which data flows can be determined without reference to a specific algorithm? Are the data flows, in some sense, the more fundamental?

The specification language described in Chapter 2 allows the use of functions as a means of abstraction. For example, the package body of Example 6.6.2 (Page 145) could alternatively be expressed as in Example 11.2.1. In the figure, the specification is replaced by a less procedural form in which each assignment is unconditional, and involves the evaluation of a function whose details are left unspecified. The precise forms of these functions do not matter, what matters are the sets of parameters they need, which determine the data precedence graph. It is a little awkward to extend the same notation to outputs, because they are generated conditionally, and have sequential structure. However, the same effect has been achieved in Example 11.2.1 by generating each output unconditionally, with the addition of a boolean parameter, which determines whether the output is truly needed. The point of Example 11.2.1 is to show that the designer does not really need to know the algorithms of events in detail in order to draw SDG's. There are several variations of Example 6.6.2 that would lead to the same behaviour, and which would all have the same SDG. For example, any local variable in Example 6.6.2 could be eliminated by textually replacing all references to it by the expression assigned to it. In a sense then, the SDG is more fundamental than the algorithm of Example 6.6.2 or the algorithm of Example 11.2.1.

On the other hand, it is hard to see how an SDG can safely be drawn without having some algorithm in mind. To do this would be know absolutely that a certain datum is needed to compute a given result, and that certain others are not. This is reasonable, for example, in the case that an employee's income tax is known to be dependent on the employee's taxable income, because the tax authorities insist on it. In a case like this, it is known that some function for computing the desired result must exist, and it is only a matter of finding out what it is. However, it may be unwise to continue the analysis without knowing more about the function concerned, because there may be additional unsuspected dependences, such as on the age or marital status of the employee.

```
package body Order_Processing is
   Authorised : array (customer) of boolean := (others => false);
   Balance, Credit_Used, Credit_Limit : array (customer) of money := (others => 0);
   Offered : array (product) of boolean := (others => false);
   Price : array (product) of money := (others => 0);
   Stock : array (product) of natural := (others => 0);
   Back_Order : array (customer, product) of natural := (others => (others => 0));
   function fn_1 (Offered : boolean) return boolean is ...
   function fn_2 (Offered, Authorised : boolean) return boolean is ...
   function fn_3 (Qty_Ordered : natural; Price : money) return money is ...
   function fn_4 (Credit_Limit, Credit_Used : money) return money is ...
   function fn_5 (Offered, Authorised : boolean; Qty_Ordered : natural;
                  Price, Credit_Limit, Credit_Used : money) return boolean is ...
   function fn_6 (Qty_Ordered, Stock : natural) return natural is ...
   function fn_7 (Offered, Authorised : boolean; Qty_Ordered, Stock : natural;
                  Price, Credit_Limit, Credit_Used : money) return boolean is ...
   function fn_8 (Offered, Authorised : boolean;
                  Back_Order, Qty_Ordered, Stock : natural;
                  Price, Credit_Limit, Credit_Used : money) return natural is ...
   function fn_9 (Offered, Authorised : boolean;
                  Qty_Ordered, Stock : natural; Balance, Price,
                  Credit_Limit, Credit_Used : money) return money is ...
   function fn_10 (Offered, Authorised : boolean;
                  Qty_Ordered, Stock : natural;
                  Price, Credit_Limit, Credit_Used : money) return natural is ...
   function fn_11 (Offered, Authorised : boolean;
                  Qty_Ordered : natural;
                  Price, Credit_Limit, Credit_Used : money) return money is ...

   procedure Order (Who : customer; What : product; Qty_Ordered : positive) is
   begin
      Error.Product (What, fn_1(Offered(What)));
      Error.Customer (Who, fn_2(Offered(What), Authorised (Who)));
      Error.Credit (Who, What, fn_3(Qty_Ordered, Price(What)),
                    fn_4(Credit_Limit (Who), Credit_Used (Who)),
                    fn_5(Offered(What), Authorised (Who), Qty_Ordered,
                       Price(What)), Credit_Limit (Who), Credit_Used (Who))));
      Invoice.Deliver (Who, What, fn_6(Qty_Ordered, Stock (What)),
                    fn_7(Offered(What), Authorised (Who), Qty_Ordered,
                       Stock(What), Price(What), Credit_Limit (Who),
                       Credit_Used (Who)));
      Back_Order (Who, What) := fn_8(Offered(What), Authorised (Who),
                    Back_Order (Who, What), Qty_Ordered, Stock (What),
                    Price(What), Credit_Limit (Who), Credit_Used (Who));
      Balance (Who) := fn_9(Offered(What), Authorised (Who),
                       Qty_Ordered, Stock (What), Balance (Who), Price(What),
                       Credit_Limit (Who), Credit_Used (Who));
      Stock (What) := fn_10(Offered(What), Authorised (Who),
                    Qty_Ordered, Stock (What),
                    Price(What), Credit_Limit (Who), Credit_Used (Who));
      Credit_Used (Who) := fn_11(Offered(What), Authorised (Who), Qty_Ordered,
                          Price(What), Credit_Limit (Who), Credit_Used (Who));
   end Order;
end Order_Processing;
```

EXAMPLE 11.2.1: SPECIFYING AN 'ORDER' WITH CREDIT USED

It is hard to believe that dependences exist independently of algorithms, for the following reason: Suppose that the astrologers are right, and everything is foretold by the stars. Then, given the time of birth of each client, it would be possible to predict their actions. In particular, it would be possible to predict exactly not only when they would place orders, but what they would order, and how much. On that basis, an order-processing system could operate with no input other than the date, and its SDG would be trivial. Although this sounds far-fetched, it is impossible to rule it out by logic. It is impossible to prove that an oracle capable of predicting any desired behaviour cannot exist, because for any given sequence of events, there is clearly at least one oracle that would have predicted them.

As a more realistic use of an oracle, consider the specification of Example 7.5.1 (Page 174), which finds each employee's salary as a percentage of total salaries. This 'clearly' requires a sequence of two loops, the first to accumulate the total, the second to compute each percentage. All the percentages depend on the total, and the total depends on all the salaries. But suppose an oracle could guess the total. Then one loop could both compute the percentages and accumulate the total. At the end of the loop, the procedure could check that the oracle's guess was actually correct. The interesting point is that such an oracle might exist in practice. It is likely that for control purposes the personnel department already knows the total payroll — probably by adjusting a previous figure according to whatever changes are made. This control figure could then serve as the required oracle.

Consequently, although it is feasible to derive dependences from algorithms, it is hard to prove that they have an independent existence.

11.2.2 Non-Procedural Specification

One reason for choosing a procedural specification language is that it allows process specifications to be derived that are also procedures. Perhaps this choice has sacrificed some other useful property. An alternative approach would be to use a non-procedural specification language, on the lines of 'Z' [Spivey 1989, Potter *et al.* 1991], for example. In 'Z', each event would be specified as a relation between a pre-condition and a post-condition. Essentially, 'Z' expresses the state of the database after an event in terms of its state before it. This has the advantage that there are no intermediate assignments to consider. Essentially there is at most one assignment per event to any given attribute. Although this simplifies dependence analysis, it only achieves the same result as is obtained by use-definition analysis; therefore it does not solve any problem that has not already been solved.

In fact, the author gave serious consideration to using 'Z', or something similar, for the specifications in this thesis. Its use was rejected for two reasons: First, it is difficult to adapt it to a process-oriented description; 'Z' does not support the notion of processes, and it provides no representation of data-flows, except between the system and its environment. Second, even if 'Z' were adapted in some way to overcome these problems, its use would be less convincing. It would lead to a set of process specifications that were non-procedural, and there would remain the mystery of how these specifications could be implemented. In contrast, the delayed procedure call mechanism used here works so smoothly that it would be a pity not to use it.

11.2.3 Eliminating States

Another potential criticism, which has already been mentioned, is the description of the system in terms of states stored in the database, rather than its external behaviour. In particular, since Section 6.6 showed how the choice of how the system state is represented can affect the efficiency of an implementation, a behaviour-based specification might eliminate the problem. The assumption here is that a behavioural specification says nothing about states, or how they are represented.

One possibility is adapt JSD's process descriptions. Although these were referred to in Section 11.1.3 as describing finite-state automata, they actually do it indirectly, by describing the grammars accepted by them. In this respect they do not define states, and there can be many automata that implement a given grammar. In this way, the same grammar corresponds to many possible state machines.

Unfortunately, the converse is also true: a given class of state machines can be described by many possible grammars. Another problem is that there is no tractable way to derive the set of all possible state machines from a grammar. There does exist a formal method of deriving a state machine from a given grammar and reducing it to minimal form, provided the grammar is regular [Hopcroft & Ullman 1979a]. This process has exponential complexity in the length of the grammar. If two regular grammars are equivalent, they lead to the same minimal state machine. This is the only certain way to discover if two grammars are equivalent, so proving that two regular grammars are equivalent is exponentially complex. In fact, the minimal state machine is a canonical form that captures the whole set of grammars with the same behaviour. Deciding the equivalence of context-free grammars is even harder, being undecidable [Hopcroft & Ullman 1979b]. There is therefore little that can be said in favour of grammar-based descriptions of behaviour in preference to state machines.

Even though there is a canonical representation for each regular grammar or finite-state machine, the states of the canonical machine may still be modelled in many ways. A set of N states may be modelled in $\log_2 N$ bits by numbering the states and recording the states as binary numbers. Unfortunately there are $N!$ ways in which the numbers 1–N can be assigned to N states. Since each bit can be modelled in a database as a separate boolean attribute, this leads to many possible representations. In addition, there are redundant representations using more than the minimum number of bits. For example, each state could be represented by a separate attribute, with only the attribute for the current state being true. Many other scenarios are possible. Even if a canonical representation of a state machine were to be defined, there is no guarantee that it would prove to be optimum.

11.3 Extensions of the Method

11.3.1 Real-time Systems

Although CDM has been applied here in the context of batch systems, it remains valid in the degenerate case when a batch contains one event. Such a situation arises in real-time systems. The separable components of a real-time system may be linked by queues, but in constast to a batch system, their queues should be a short as possible, serving only to buffer events that a downstream process has been too slow to serve. In some cases the components may be separated by as little as a procedure call, or even merged into a single process. In these cases, separability offers no performance benefit, but serves only to modularise the system into smaller processes that can be separately tested and understood. In other cases, it may be desired to communicate between the processes *within* a strong component. This may be because of external constraints: for example, the processes may need to be assigned to physically separate processors, or it may be done purely for modularity.

Within a strong component, communication must be two-way. Before it updates its own state, each process must communicate its state to other processes and discover the states of the processes on which it depends. There are typically many ways this can be done. The simplest is to arrange the processes as a sequence. Each process except the last calls the next using remote procedure call, passing it its own initial state, plus any initial states passed to it by its caller. This guarantees that the last process in the chain has access to all the state variables in the component. Each process returns from the procedure call its own initial state, plus any initial states returned by the process it has called. Thus every process has access to all initial state information, both from earlier and later processes in the sequence and is ready to update its state. If communication costs are considered, such a linear arrangement may not be optimal. Choosing the optimal communication pattern is not considered in this thesis.

To illustrate how CDM can be applied to a real-time system, consider the guidance sub-system of a surface-to-air missile. (The missile *system* also includes aerodynamic, propulsion and weapons sub-systems.) The purpose of the guidance sub-system is to convert positional and motion information into signals that operate servo-motors.

Two servo-motors control the direction of a radar dish, helping it remain pointing at the target. Four other servo-motors adjust the angles of control surfaces arranged at 90° intervals around the axis of the missile, labelled ‘N’, ‘S’, ‘E’ and ‘W’ in Figure 11.3.1. Through aerodynamic forces, they control rotation of the missile about any of its three axes. Rotation about the long axis of the missile is called ‘roll’, rotation about its nominally vertical axis is called ‘yaw’, and rotation about its nominally horizontal axis is called ‘pitch’. Coordinated motion of the ‘N’ and ‘S’ control surfaces induces yaw, coordinated motion of the ‘E’ and ‘W’ surfaces induces pitch, and differential motion of all four surfaces induces roll.

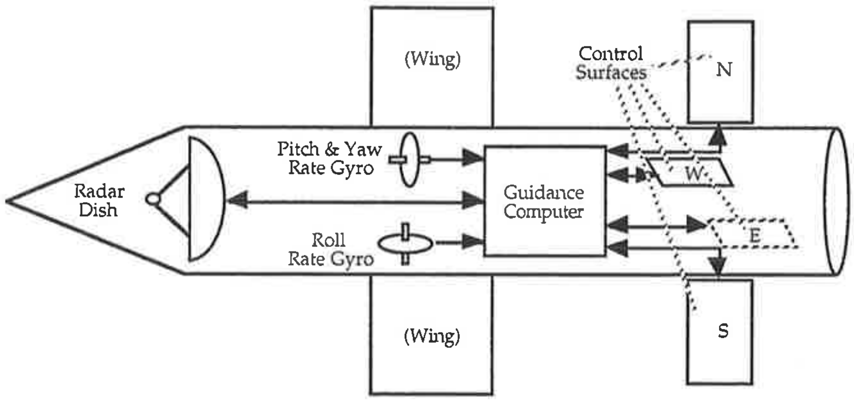

FIGURE 11.3.1: A MISSILE GUIDANCE SUB-SYSTEM

The aims of the guidance sub-system are to prevent the missile from rolling, and to adjust the pitch and yaw rates of the missile so as to keep the pitch and yaw angles of the radar dish constant. These conditions will ensure that the missile maintains a constant bearing with the target, a property of a collision course.

A gyroscope mounted with its axle aligned with the long axis of the missile measures its rates of pitch and yaw, and another with its axle at right angles to the long axis measures the missile's rate of roll. These are *rate* gyroscopes: they do not remain at the same orientation in space, instead, they remain at the same orientation with respect to the missile, and the forces with which they resist precession measure rotational velocities. The target radar measures the pitch and yaw deviations of the target from the axis of the radar dish. Two further sensors measure the pitch and yaw of the dish from the long axis of the missile. Finally, four sensors measure the angles of the control surfaces relative to the same axis. All these inputs are processed by the guidance computer to determine the control signals sent to the six servo-motors. We assume that the inputs are sampled at regular intervals. Each sample constitutes an event.

Figure 11.3.1 suggests that, apart from the gyroscopes, the whole system forms a single strongly connected component: data flows both to and from the radar and the control surfaces. In this respect the figure is misleading, because data flows from a *detector* and angle *sensors* mounted on the radar dish and to its *servo-motors*, and from *sensors* on the control surfaces and to their *servo-motors*. From the point of view of the control system, the detector and sensor signals are inputs and the servo-motor signals are outputs. Because power to a servo-motor eventually results in motion of the target sensor or a control surface, which then changes the signal coming from a detector, there is a feedback effect.

To apply the CDM method to this system, we must consider its state variables. These are the pitch and yaw rotations of the radar dish, and the positions of the control surfaces. We assume that the radar dish servo signals depend on the radar detector outputs, the dish's current position, and its rate of movement. We also assume that the desired control surface positions are determined by the rates at which the radar dish moves, and the signals from the rate gyroscopes. Finally, the control servo signals depend on their desired positions, their measured positions, and their rates of motion.

It may surprise the reader that there are any state variables at all. For example, the state of the 'N' control surface can surely be found by reading the signal from its angle sensor. However, to adequately control the surface, it is not enough to know its position, one must also know its velocity: If it is in the correct position and at rest, no signal should be sent to its servo-motor, but if it is in the correct position and moving, a signal needs to be sent to stop its motion and return it to the correct position. The velocity of the surface can estimated from the difference between its old position and its new position, measured over the time between events. The same reasoning applies to the other three control surfaces and to the pitch and yaw motions of the radar dish.

Through aerodynamic forces, the control surfaces ultimately determine the position of the missile in space, and therefore the signals reaching the radar detector. The missile's environment creates a feedback loop. Therefore the exact rules by which the guidance sub-system computes its outputs from its inputs must change the missile's orientation in such a way that the feedback does cause instability. Although the analysis of this feedback is an important part of the design of the control rules, it lies outside the ambit of the design problem considered here. However, feedback via the environment not a unique feature of missile systems. When an order entry system sends goods in response to customer orders, its outputs influence the future actions of customers, completing a feedback loop. Such dynamic behaviour of business systems is thought to cause trade cycles.

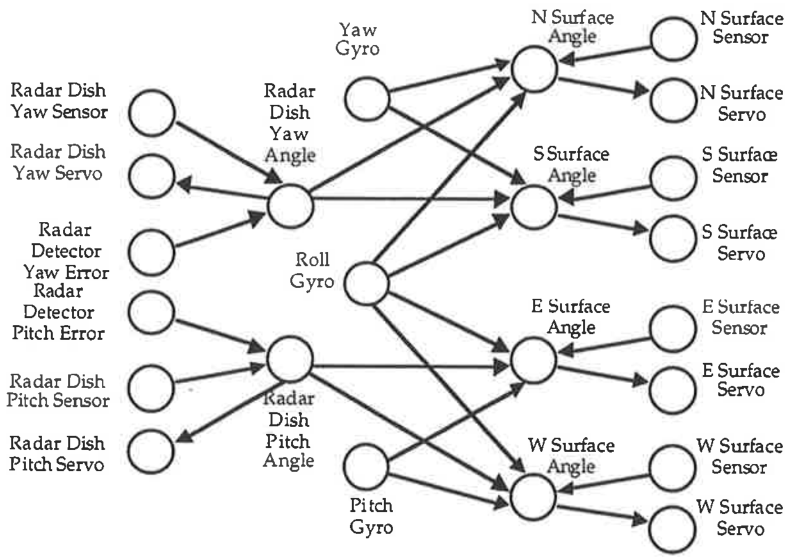

FIGURE 11.3.2: THE MISSILE GUIDANCE SDG

Figure 11.3.2 shows the SDG of the guidance sub-system, including its inputs and outputs, for a hypothetical set of control rules.

Inspection of the SDG of Figure 11.3.2 reveals the familiar property of separability: The control surface angles depend on the radar dish angles, but not the reverse. It is also reveals the property of independence: The two radar dish angles are independent of one another, and the four control surface angles are independent of one another. This suggests a decomposition into two similar radar dish controllers and four similar control surface controllers. The advantages of such a decomposition are that the system can be made from replicated parts and tested in modular fashion.

It will thus be seen that, although the objectives of system decomposition may differ, the CDM approach can be used to decompose dynamic systems of all kinds, and is not restricted to the analysis of batch systems.

On the other hand, having shown that the CDM method can be applied to the design of a missile guidance sub-system, we now have to admit that in such a context it is almost worthless. The reason is simple: Every missile guidance system has much the same functional requirement, and therefore much the same structure. A design similar to that suggested by Figure 11.3.2 has been used in virtually every missile ever built, and CDM has nothing new to tell us. For CDM to be useful, the problem must have some novel structure. Further, for CDM to yield a helpful result, the structure should not be too richly interconnected, or no decomposition will be possible. Business information systems are a fruitful source of suitable problems, but there may not be many others.

11.3.2 Avoiding Deadlock in Databases

This section is a footnote to Section 3.6, which discussed a contention-free parallel database. It was pointed out that if processors were allocated to events without first decomposing them into their minimal separable processes, the processors would have to contend for rows in the database. This would necessitate the use of a locking protocol to prevent unwanted interactions between transactions. In the two-phase protocol, when a transaction reads a row or wishes to update a row, it locks it. The transaction retains the lock until it is complete, when it releases all its locks. When a transaction wants to read a row, it locks it in shared mode: other transactions may read the row. When a transaction wants to update a row, it locks it in exclusive mode: no other transaction may access the row at all. This prevents two updates from interacting, and prevents rows from being inspected when they in the process of being updated, but does not prevent read-only access by several transactions at once.

An unfortunate property of locking protocols is that they can cause deadlock. For example, transaction T may have locked row R and wish to access row S. It is forbidden from doing this because transaction U has already locked row S. Once transaction U releases the lock on row S, transaction T can continue. Sadly, transaction U wishes to access row R and is waiting for transaction T to release it. This situation is called deadlock. It is only possible to remove the deadlock by one or more transactions relinquishing their rights to the records they have locked. Typically, the transactions are aborted, and started again. This is a nuisance in two ways: first, the database management system must have the means to prevent aborted transactions from changing the database, and second, it may inconvenience the system's users. Deadlock can be detected by drawing a graph in which each vertex represents a row. An edge is drawn from row R to row S if some transaction T has locked row R and wants to access (and lock) row S. The edge is labelled 'T', to identify the transaction that caused it. If two or more vertices are strongly connected, they are deadlocked. Deadlocks can be efficiently detected by depth-first search of such a graph.

It turns out that deadlock can never occur in the Macrotopian Library database system: the deadlock graph can never contain a cycle. The update events, such as 'Borrow', first access a book.row, then a branch row. All the edges they generate lead from book rows to branch rows, and cannot cause cycles. The only possibility is for cycles to be present within the book rows or within the branch rows. These could only be caused by 'Audit' events, but these only used shared locks, so they cannot cause deadlock either.

Is it possible that some adaptation of the techniques described here could be used to design systems that, although unable to avoid contention, are deadlock free? This would be useful when a batch design is impossible or inappropriate. Being certain that deadlock could not occur would be a major advantage. It would ensure that every transaction could commit successfully, and would remove the need for undoing uncommitted updates. This would reduce database management overhead and simplify database recovery.

First, we note that if every database attribute can be given a rank, and attributes are only ever accessed and locked in rank order, then deadlock cannot occur. This can be seen by considering the graph whose vertices are database rows, a cycle in which indicates a deadlock. If the rows are totally ordered, and they are accessed in rank order, a cycle can never occur. The question is, can a suitable ordering be imposed on the rows?

In fact, by ranking tables and rows, it is *always* possible to avoid deadlock. The default strategy is to lock the whole of every table, in rank order. This will rarely be the optimum strategy, so the question of choosing the best ranking is an optimisation problem. Despite this, it is surprising that this approach to deadlock-free locking does not appear to have been previously studied.

This is not quite the same problem as the one we have already considered. For example, if row A depends on B, it cannot be updated until row B has been locked, but there is no reason why row A cannot have been locked before row B. The only requirement is that the program must know that row A needs to be locked before it locks row B. The only way that it could fail to know this is when row B determines which A row is selected. For example, B could be an enrolment, and A could be the student who made the enrolment. Suppose students is ranked before enrolments. In this case, to avoid deadlock, it would be necessary to lock all the student rows. Since this is inefficient, a better strategy would be to rank enrolments before students. The need for this ranking arises because an enrolment contains a foreign key that references a student, and may therefore be part of an access path that is used to find the student. Assuming that access paths typically follow foreign key – primary key connections, heuristically, a good ranking may be discovered by drawing a graph whose vertices are tables and whose edges represent links from foreign keys to primary keys, i.e., from children to their parents. If the resulting graph is acyclic — and it often is — then each topological sort of the graph gives a possible ranking of the tables.

In addition to ranking the tables, it is also necessary to rank the rows within a table. consider the example of a 'Safe Transfer' between accounts considered in Example 5.2.3. Here, if the 'Payer' account is always locked before the 'Payee' account, deadlock may occur. For example, two transactions might take place concurrently in which the roles of payer and payee were interchanged. But if the row with the smaller account number were always locked before the row with the higher account number, irrespective of its role, deadlock would be avoided. In other words, rows within a table ought to be ranked, just as tables should be.

Of course, it is possible to reverse the usual direction of access, from child to parent, and retrieve a parent before its children. Assuming that the children must be locked first, this may mean locking the entire child table. But if the database system supports hierarchical locking, it may be possible to lock only the child rows concerned, provided that the parent key has been used to cluster the rows. Of course, if a child table has more than one parent table, only one parent can control the clustering. For example, an enrolment has two parents: a student, and a subject. Hierarchical locking allows that, if the enrolments are clustered by subject, then to access the enrolments for a subject needs only one cluster to be locked, but to access the enrolments for a given student needs the entire table to be locked. Conversely, if the enrolments were clustered by student, then to access the enrolments for a given subject would need the entire table to be locked, but to access the enrolments for a student would need only one cluster to be locked.

Sometimes it is not possible to access the rows of a table in rank order. For example, a Bill of Materials database involves an undirected cycle in which an assembly comprises several components. Each component may itself be an assembly, recursively. Ultimately, all assemblies comprise a set of basic parts. Although the same component may be used by several assemblies the parent-child graph is necessarily acyclic. It is possible to rank the assemblies by level, so that working from higher levels to lower levels locks rows in rank order. Unfortunately, when it is necessary to work from lower levels to higher levels, the rows must be accessed in the reverse of rank order, so it becomes necessary to lock the whole table.

There is always a deadlock-free implementation of any system, and the above examples suggest that deadlock-free systems may be a practical solution in many instances. In so far as they can implement a wider range of specifications that contention-free systems can, their study may prove worthwhile.

11.4 Contributions of the Thesis

In this section, the author wishes to formally state what he believes are the original contributions made by the thesis. Of course, these claims are always 'to the best of the author's knowledge.' The discovery of single example proves that something exists, but any number of absences of an example can't prove that it doesn't — especially when it may exist in some unrelated area of computer science, using different terminology.

11.4.1 Generalisation of Sequential Update Algorithm

Although correct descriptions of the sequential update algorithm have appeared previously [Dijkstra 1976, Dwyer 1981a], they have assumed that all events affect at most one master record, and that reports are only produced concerning the final state of the master file. In this thesis, we have described how to implement events that affect many master records, and how to report the state of the master file at any desired time.

11.4.2 Parallel Batch Processing

The parallel processing algorithm of Section 3.4 is an analogue of the sequential access batch system described in Section 3.3. This algorithm was described by the author in an earlier report [Dwyer 1995]. It is believed to be original. Also, although contention-free databases may not be new, a systematic method of deriving their design is new.

11.4.3 Real-Time Equivalence

The author claims that the idea of real-time equivalence presented in this thesis is the first formal attempt to define what it means for the design of a batch system to be correct.

11.4.4 Separability

Separability is an old idea, but its importance in system design has gone unrecognised — or perhaps it is truer to say it has only been used informally. The formal use of dependence analysis to derive the separable processes of a system as the strongly-connected components of an SDG has not previously been reported.

11.4.5 Independence

Independence is a trivial idea whose importance has long been implicitly recognised in designing parallel algorithms. Where this thesis makes a new contribution is in generalising its definition, making it possible, even when it is heavily disguised, to detect the *potential* for parallelism by first decomposing a system into separable parts. Independence becomes then both the motive for decomposition, and a criterion for constraining optimisation.

11.4.6 Extensions to Use-Definition Analysis

The use-definition analysis used in this thesis extends conventional techniques in two ways: First, it distinguishes array elements, so that it becomes possible to test for independence. Second, it extends the usual idea of lexical definitions to that of dynamic definitions, making it possible to test for independence in specifications containing loops.

11.4.7 The State Dependence Graph

There are really two kinds of SDG described here: the kind used by the *Designer* program, and the more informal kind suitable for use by a human systems analyst. It is the second kind which is of interest here. Essentially, it provides a formal way for an analyst to determine the CPG of a system. By colouring vertices and marking edges, the notion of compatibility also provides a semi-formal way of helping the designer optimise the process graph. As discussed in Chapter 6, it also provides a useful tool with which the analyst may consider design options and quickly assess the implications of a given specification. As such, it may assist the analyst in negotiating a new specification with the client that can be implemented more efficiently. The author believes that the use of these graphs is original.

11.4.8 System Design as a Composition Problem

A major contribution of this thesis is that instead of system design being perceived as a problem in optimal decomposition, it becomes one of a canonical decomposition followed by optimal composition. This insight serves to greatly reduce the search space of the optimisation problem, and to help direct the progress of optimisation.

11.4.9 System Optimisation as a Lattice Problem

Section 9.2 demonstrates that the set of feasible system designs may be expressed as a lattice. The author believes that this is the first time the problem has been expressed in this way. The major benefit of this insight is to see that pair-wise composition is all that is needed to reach an optimum.

11.4.10 NP-Completeness of Process Optimisation

Section 9.6 proves that the composition problem can be NP-complete. The author believes that this strongly suggests that finding the optimal process graph from a specification is also NP-complete. It does not prove it, however. It is possible that a radically different approach might not involve the composition problem, although it seems unlikely.

11.4.11 Optimisation Heuristics

The optimisation heuristics described in Subsection 10.5.3 have proved successful in the case of the examples given in this thesis, and are a useful contribution to the practical problem of automating system design.

11.4.12 The Designer Program

The *Designer* program demonstrates that it is feasible to automate system design. Although previous authors have described similar programs [Teichroew & Sayani 1971, Nunamaker *et* *al.* 1976, Teichroew & Hershey 1977], they had narrower scope than the program described here.

11.4.13 A Design Methodology

Finally, the author claims that the synthesis of all the ideas in this thesis form a new design methodology, which should prove of great value to systems analysts and designers. Many practicing systems analysts have been surprised at some consequences of CDM, although it must be admitted that not all of them are willing to accept real-time equivalence as the criterion of correctness. It seems to be a common belief that any system requirement can be implemented using sorting and sequential access, without the need for random access. On the author demonstrating that for some problems random access is necessary to preserve real-time equivalence, real-time equivalence has been rejected as too strict. On the other hand, no-one has yet suggested an alternative to it that is both useable and less strict.

The author also claims that the method has a very wide range of applicability, as exemplified by the case studies of Sections 5.6 and 11.3.1, although, because many problems are already well solved, it has a much narrower range of utility.

11.5 Some Frank Remarks

Someone once said that research is like entering a darkened room and finding the right switch. Once the lights are on, everyone can see. Although the ideas in this thesis make a coherent story, getting them to hang together has not been easy.

Although the author soon recognised that the strong components of the SDG determined a canonical process graph, it was not easy to get the details correct. On the one hand, early forms of dependency analysis could lead to CPG's that had no obvious implementation, or they might fail to find valid CPG's that did. These problems had to be remedied by generalising the known update algorithms or by adjusting the rules for defining dependences. For example, the 'Careful Transfer' of Example 5.2.5 caused such a problem. Its CPG suggested that two independent access processes would implement the specification efficiently, but it took the author a long time to see how to add preprocess and postprocess phases to the usual update algorithm so that it could handle it correctly.

Another major source of problems was detecting independence. This is so easy to do intuitively that the author was amazed by the complexity involved in automating it. In retrospect, the ease with which independence can be found intuitively is often based on external knowledge: in an order processing system, we don't expect different products or customers to interact. It is not so easy to detect from the text of the specification alone. The author had also hoped to find that the problem had already been solved by previous research into parallel algorithms, but never succeeded in finding any work that was relevant.

Similarly, the analysis of independence caused embarrassment, in that the graphical approach often showed the possibility of independent access without suggesting how to implement it. An example of this is loop folding, as described in connection with Example 7.5.4. Again, a major problem was to find a set of rules for defining independence that would correctly match the independent update algorithms and the text rewriting rules for delayed procedure call. Of course, such difficulties really mean that additional program transformations — such as loop folding — would improve and generalise the results presented here, but every piece of research has to stop somewhere.

At one point, when it became apparent that lexical definitions were inadequate, they were entirely abandoned in favour of dynamic definitions. But this often gave rise to designs where one assignment statement was assigned to two different processes. In fact, this was a clue that a loop could be unfolded into a series of separate processes, along the lines of the iterative approach to the Bill of Materials Problem of Section 5.6.

Part of the problem of analysing independence was to construct the SDG correctly — including the dependencies for inspections and updates of arrays and their elements. Finding the correct constructions proved very elusive, because the SDG must serve two purposes, to determine separability, and to determine independence. The notion of dynamic definitions proved useful here. Even so, it seemed very hard to get one property right without spoiling the other. In fact, a case was eventually found in which including a particular edge gave the correct analysis of separability, but the same edge had to be omitted to derive the independence information. *Eventually*, this led to the decision to classify the edges of the SDG as hard or soft.

Another problem was to choose the right form for a specification. Because SASS and JSD both ignore the details of specification algorithms and concentrate on data flow, it took the author a long time to see that data flow is not a given, but is ultimately based on an algorithm, even if the algorithm is never formalised. Once this had been realised, the advantages of a procedural specification over a precondition-postcondition style of specification were quickly realised, including the possibility of automatically deriving process specifications.

Even so, the author's first attempts at formal specifications were a failure, being based on 'Z', where system outputs are represented by output variables. This form of specification failed to model process communications within the system properly, especially the case where one input produces multiple outputs. Other attempts were based on process algebras, such as CSP. These proved too general, and the burden of proving the correctness of a design was too complex. The idea of modelling process communication by delayed procedure call may seem obvious now, but it didn't seem obvious then. In fact, even after the author had decided to model outputs of internal processes by procedure calls, for some reason the logical step of specifying the whole system in the same way took a little longer. This stage of development can be seen in the author's earlier technical report [Dwyer 1992].

A further hang up concerned the NP-completeness of the optimisation problem. To the author, it was perfectly obvious that the problem was NP-complete, but that was not the same as proving it. Clearly, if the thesis could not show how to solve the optimisation problem efficiently in all cases, then it must prove that it can't be done — at least as far as anyone knows. The method of reducing a known NP-complete problem to a new one is well known, but it was not easy to spot what known problem would provide the proof. Also in this context, it was 'obvious' that feasible pair-wise compositions were all that needed to be considered, but proving it was somewhat harder.

In this respect, it may be added that actually implementing the *Optimiser* has suggested that the author's faith in feasible pair-wise compositions may be misplaced. As mentioned earlier, composing pairs of state variables is what matters, and state variables may be connected by long paths in the full SDG. Usually, a long series of pair-wise compositions has to succeed before two state variables are composed. A faster method might be to proceed more on the lines of the manual method, and attempt to compose pairs of system variables, whether their compositions are feasible or not. This approach has yet to be tested.

To complete this litany of failure, the author would like to confess that many problems were revealed only when — after false starts in Pascal and Ada — attempts were made to implement the *Designer* program. Arguments that seemed convincing on paper failed to stand up to experiment. Conversely, early prototypes of the program sometimes stood up well to experiment, but were revealed to be faulty when an attempt was made to explain them on paper. They were easily proved to be incorrect by counter-examples, many of which are now immortalised in this text.

12. References

Aguilar 1996 J. Aguilar, “Heuristics to Optimize the Speed-Up of Parallel Programs”, *Lecture Notes in Computer Science **1127**: Parallel Computation: Proceedings 3rd International ACPC Conference, Klagenfurt*, 174–183, (L. Böszörményi *ed.*), Springer (1996).

Aho & Ullman 1972a A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing*, Ch. 2, Prentice-Hall, Englewood Cliffs, NJ (1972).

Aho & Ullman 1972b A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation, and Compiling, Volume 2: Compiling*, Ch. 11, Prentice-Hall, Englewood Cliffs, NJ (1972).

Aho *et al.* 1972 A.V. Aho, M.R. Garey and J.D. Ullman, “The Transitive Reduction of a Directed Graph”, *Society for Industrial and Applied Mathematics Journal of Computing*, **1**, 131–137, (1972).

Aho *et al.* 1986 A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* Ch. 10, Addison-Wesley, Reading, MA (1986).

Andrews 1981 G.R. Andrews, “Synchronizing Resources”, *ACM Transactions on Programming Languages and Systems,* **3** (4), 405–430 (Oct 1981).

Bacon *et al.* 1994 D.F. Bacon, S.L. Graham and O.J. Sharp, “Compiler Transformations for High-Performance Computing”, *ACM Computing Surveys,* **26 (4)**, 345–420, (Dec 1994).

Barnes 1989 J.G.P. Barnes, *Programming in Ada,*. Addison Wesley, Reading, MA (1989).

Bernstein *et al.* 1986 P.A. Bernstein, N. Goodman & V. Hadzilacos, *Concurrency Control and Recovery in Database Systems,*. Addison Wesley, Reading, Mass. (1986).

Bertziss 1986 A. Bertziss, “The Set-function Approach to Conceptual Modelling”, *Information Systems Design Methodologies: Improving the Practice*, 289–318, (Olle, T.W., Sol, H.G. and Verijn-Stuart, A.A. *eds.*), North-Holland, Amsterdam (1986).

Bidoit & Amo 1995 N. Bidoit and S. De Amo, “A First Step Towards Implementing Dynamic Algebraic Dependencies”, *Lecture Notes in Computer Science **893**: Database Theory: Proceedings ICDT ’95*, 308–319, (G. Gottlob & M.Y. Vardi *eds.*), Springer (1995).

Booch 1991 G. Booch, *Object-oriented Design with Applications*, Benjamin-Cummings, Redwood City, Calif. (1991).

Brady 1997 M. Brady, *Open Prolog Home Page*, http://www.cs.tcd.ie/open-prolog/, Trinity College, Dublin. (1997).

Bubenko 1986 J.A. Bubenko, "Information System Methodologies — a Research View", *Information Systems Design Methodologies: Improving the Practice*, 289–318, (Olle, T.W., Sol, H.G. and Verijn-Stuart, A.A. *eds.*), North-Holland, Amsterdam (1986).

Cameron 1983a J.R. Cameron, *JSP & JSD : The Jackson Approach to Software Development*, IEEE Computer Society Press, Silver Spring MD, (1983).

Cameron 1983b J.R. Cameron, 203–212, "Two Pairs of Examples in the Jackson Approach to System Development".

Cameron 1986 J.R. Cameron, "An Overview of JSD', *IEEE Transactions on Software Engineering,* **12**(2), (Feb 1986).

Casati *et al.* 1995 F. Casati, S. Ceri, P. Pernici, G. Pozzi, "Conceptual modelling of Workflows", *Lecture Notes in Computer Science 1021: OOER '95: Object-oriented and Entity-Relationship modeling*, 341–354, (M.P. Papazoglou *ed.*), Springer (1995).

Chandy & Misra 1988 K.M. Chandy & J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading Mass. (1988).

Chapin 1981 N. Chapin, "Structured Analysis and Structured Design: an Overview", *Systems Analysis and Design: A Foundation for the 1980's*, 199–212, Elsevier North Holland, New York (1981).

Chen 1976 P.P. Chen, "The Entity-Relationship Model — toward a Unified View of Data", *ACM Transactions on Database Systems*, **1**(1), 9–36, (Mar 1976).

Clocksin & Mellish 1984 W.F. Clocksin & C.S. Mellish, *Programming in Prolog* , Springer, New York. (1984).

Codd 1970 E.F. Codd, "A Relational Model of Data for large shared Data Banks", *Communications of the ACM*, **13**(6) 377–387, (Jun 1970).

Colter 1982 M.A. Colter, "Evolution of the Structured Methodologies", *Advanced System Development/Feasibility Techniques*, Couger, J.D., Colter, M.A., and Knapp, R.W. (eds.), Wiley, New York (1982).

Connor 1981 M. Connor, "Structured Analysis and Design Technique", *Systems Analysis and Design: A Foundation for the 1980's*, 291–320, Elsevier North Holland, New York (1981).

Conway 1963 M.E. Conway, "Design of a Separable Transition-diagram Compiler", *Communications of the ACM*, 6(7) (July 1963).

Date 1993 C.J. Date, *A Guide to the SQL Standard*, Addison-Wesley, Reading Mass. (1993).

Delisle *et al.* 1982 N.M. Delisle, D.E. Menicosy, and N.L. Kerth, "Tools for supporting Structured Analysis", *Automated Tools for Information System Design*, Schneider, H.-J., and Wasserman, A.I. (eds.), 11–20, North-Holland, Amsterdam (1982).

DeMarco 1978 T. DeMarco, *Structured Analysis and System Specification*, 27–31, Yourdon Press (1978).

Devirmis & Ulusoy 1996 T. Devirmis and Ö. Ulusoy, "A Transaction Model for Multidatabase Systems", *Lecture Notes in Computer Science **1124**: Euro-Par'96 Parallel Processing: Proceedings Volume II*, 862–865, (L. Bougé, P. Fraigniaud, A. Mignotte & Y. Robert *eds.*), Springer (1996).

Dijkstra 1968 E.W. Dijkstra, "The Structure of the THE Multi-programming System", *Communications of the ACM*, **11**(5), 341–346, (May 1968).

Dijkstra 1976 E.W. Dijkstra, *A Discipline of Programming*, Ch. 15, Prentice Hall, Englewood Cliffs (1976).

Dwyer 1981a B. Dwyer, "One more Time — How to update a Master File", *Communications of the ACM*, **24**(1), 3–8, (Jan 1981).

Dwyer 1981b B. Dwyer, Author's Response to Technical Correspondence, *Communications of the ACM*, **24**(8), 538–539 (Aug 1981)

Dwyer 1992 B. Dwyer, "The Data Dependency Method of Systems Design", *Computer Science Technical Report 92-05*, University of Adelaide, 1992.

Dwyer 1995 B. Dwyer, "Contention-free Scalable Parallel Batch Processing: Exploiting Separability and Independence", *Technical Report TR95-03,* Department of Computer Science, University of Adelaide (1995).

Dwyer 1998 B. Dwyer, "Separability Analysis Can Expose Parallelism", *Proceedings of PART '98: The 5th Australasian Conference on Parallel and Real-Time Systems*, 365–373, (K.A. Hawick & H.A. James, *eds.*) Springer, Singapore (1998).

Eich 1992, M.H. Eich, "Main Memory Databases: Current and Future Research Issues", *IEEE Transactions on Knowledge and Data Engineering*, 506–508, **4**(6), Dec 1992.

Eve & Kurki-Suonio 1977 J. Eve and R. Kurki-Suonio, "On computing the Transitive Closure of a Relation", *Acta Informatica*, **8**(4), 303–314 (1977).

Flores *et al.* 1993 F. Flores, M. Graves, B. Hartfield, and T. Winograd, "Computer Systems and the Design of Organizational Interaction", *Readings in Groupware and Computer Supported Cooperative Work*, 504–513, (Baecker, R.M. *ed.*), Morgan Kaufmann, San Mateo (1993).

Floyd 1986 C. Floyd, "A Comparative Evaluation of System Development Methods", *Information Systems Design Methodologies: Improving the Practice*, 19–54, (Olle, T.W., Sol, H.G. and Verijn-Stuart, A.A. *eds.*), North-Holland, Amsterdam (1986).

Frankel 1979 R.E. Frankel, "FQL — The Design and Implementation of a Functional Database Query Language", *Technical Report 79-05-13*, Dept. of Decision Sciences, University of Pennsylvania, Philadelphia. (1979).

Friberg 1984 J. Friberg, "Numbers & Measures in the Earliest Written Records", *Scientific American*, **250**(2) 78–85. (Feb 1984).

Gabber 1990 E. Gabber, "Developing a Portable Parallelizing Pascal Compiler in Prolog", *The Practice of Prolog*, 109–136, (L. Sterling *ed.*), MIT Press, Cambridge, Mass. (1990).

Gane & Sarson 1979 C. Gane, and T. Sarson, *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, Englewood Cliffs, NJ. (1979).

Garcia-Molina & Salem 1992 H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview," *IEEE Transactions on Knowledge and Data Engineering*, **4**(6), 509–516 (Dec. 1992).

Garey & Johnson 1979 M.R Garey, and D.S. Johnson, *Computers and Intractability*, Appendix A4.2[SR8] "Compression and Representation", p.228, W.H Freeman, San Francisco, CA. (1979).

Grant 1987, J. Grant, *Logical Introduction to Databases*, Ch. 5, Harcourt Brace Jovanovich, Orlando, Florida (1987).

Gray & Reuter 1992 J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1992).

Gray 1981 J. Gray, "The Transaction Concept: Virtues and Limitations," *Proceedings of the International Conference on Very Large Data Bases,* pages 144–154 (1981).

Haerder & Reuter 1983, T. Haerder & A. Reuter "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, Vol. 15, No. 4, Dec. 1983 pp. 287–317.

Hawryszkiewycz 1994 I.T. Hawryszkiewycz, *Introduction to Systems Analysis and Design*, Ch. 7, Prentice-Hall, Englewood Cliffs, NJ (1994).

Hillis 1985 W.D. Hillis, *The Connection Machine*, MIT Press (1985).

Hoare 1974 C.A.R. Hoare, "Monitors: an Operating System Structuring Concept", *Communications of the ACM*, **17**(10), 549–557 (Oct 1974).

Hoare 1978 C.A.R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, **21**(8), 666–677 (Aug 1978).

Hoare 1985 C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, Englewood Cliffs, NJ (1985).

Hopcroft & Ullman 1979a J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Ch. 3, Addison-Wesley, Reading, MA (1972).

Hopcroft & Ullman 1979b J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Ch. 11, Addison-Wesley, Reading, MA (1972).

Inglis 1981 J. Inglis, "Updating a Master File — Yet one more Time", *Communications of the ACM*, **24**(5), 299 (May 1981).

Jackson 1975 M.A. Jackson, *Principles of Program Design*, Academic Press, London (1975).

Jackson 1978 M.A. Jackson, "Information systems: Modelling, Sequencing and Transformations", *Third International Conference on Software Engineering* (1978).

Jackson 1981 M.A. Jackson, "Some Principles underlying a System Development Method", *Systems Analysis and Design: A Foundation for the 1980's*, 185–198, Elsevier North Holland, New York (1981).

Jackson 1983 M.A. Jackson, *System Development*, Prentice Hall, Englewood Cliffs, NJ (1983).

Kennedy 1981 K. Kennedy, "A Survey of Data Flow Analysis Techniques", *Program Flow Analysis: Theory and Applications,* (S.S. Muchnick and N.D. Jones *eds.*) Prentice-Hall, Englewood Cliffs, NJ (1981).

Krishnamurthy & Murthy 1991 E.V. Krishnamurthy and V.K. Murthy, *Transaction Processing Systems*, Prentice Hall, Sydney (1991).

Kobol 1987 P. Kobol, "Dining Philosophers — An Exercise in using the JSD", *ACM SIGSOFT Software Engineering Notes* **12**(4), 27–33 (Oct 1987).

Kwong & Majumdar 1996 P. Kwong and S. Majumdar, "Study of data Distribution Strategies for Parallel I/O Management", *Lecture Notes in Computer Science 1127: Parallel Computation: Proceedings 3rd International ACPC Conference, Klagenfurt*, 32–48, (L. Böszörményi *ed.*), Springer (1996).

Lampson & Sturgis 1976 B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," *Technical Report,* Computer Science Laboratory, Xerox, Palo Alto, CA (1976).

Maiocchi 1985 M. Maiocchi., "The use of Petri Nets in Requirements and Specification", *System Description Methodologies*, Teichroew, D, and Dávid, G. (Eds.) 253–274, Elsevier Science Publishers, 1985.

Malone & Crowston 1993 T.W. Malone, and K. Crowston, "What is Coordination Theory and How can it help design Cooperative Work Systems?", *Readings in Groupware and Computer Supported Cooperative Work*, 504–513, (R.M. Baecker *ed.*), Morgan Kaufmann, San Mateo (1993).

Mannino 1987 P. Mannino, "A Presentation and Comparison of Four Information Systems Development Methodologies", *ACM SIGSOFT Software Engineering Notes* **12**(2), 26–29 (Apr 1987).

Masiero & Germano 1988 P.C. Masiero, and F.S.R. Germano, "JSD as an Object Oriented Design Method", *ACM SIGSOFT Software Engineering Notes* **13**(3), 22–23 (Jul 1988).

Nunamaker & Konsynski 1981 J.F. Nunamaker, and B. Konsynski, "Formal and Automated Techniques of Systems Analysis and Design", *Systems Analysis and Design: A Foundation for the 1980's*, 213–234, Elsevier North Holland, New York (1981).

Nunamaker 1971 J.F. Nunamaker, "A Methodology for the Design and Optimization of Information Processing Systems", *AFIPS Conference Proceedings*, **38**, 1971 Spring Joint Computer Conference, May 1971, 283–294.

Nunamaker *et al.* 1976. J.F. Nunamaker, T. Ho, B. Konsynski, and C. Singer, "Computer-aided Analysis and Design of Information Systems", *Communications of the ACM*, **19**(12), (Dec 1976).

O'Keefe 1990 R.A. O'Keefe, *The Craft of Prolog,* MIT Press, Cambridge, Mass. (1990)

Olle *et al.* 1991 T.W. Olle, J. Hagelstein, I.G. Macdonald, C. Rolland, H.G. Sol, F.J.M. Van Assche, A.A. Verrijn-Stuart, *Information Systems Methodologies: A Framework for Understanding,* Addison-Wesley, New York (1991).

Potter *et al.* 1991 B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z,* Prentice Hall, London (1991)

Preuner & Schefl 1998 G. Preuner and M. Schefl, "Observation Consistent Integration of Views of Object Life-Cycles", *Lecture Notes in Computer Science* **1405**: *Advances in Databases: Proceedings BNCOD 16*, 32–48, (S.M. Embury, N. J. Fiddian, W.A. Gray & A.C. Jones *eds.*), Springer (1998).

Richter 1986 C. Richter, "An Assessment of Structured Analysis and Structured Design", *ACM SIGSOFT Software Engineering Notes* **11**(4), 75–83 (Aug 1986).

Saade & Wallace 1995 H. Saade, and A. Wallace., "Cobol '97: A Status Report", *Dr. Dobb's Journal*, 52–54 (Oct 1995).

Schneider & Lamport 1982 F.B. Schneider, and L. Lamport, "Paradigms for distributed programs", *Lecture Notes in Computer Science* **190**: *Distributed Systems*, 431–480, Springer, New York (1982).

Simpson 1986 H. Simpson, "The Mascot method", *Software Engineering Journal*, **1**(3), 103–120 (May 1986).

Smith & Barnes 1987 P.D. Smith and G.M. Barnes, *Files and Databases: an introduction*, 214, Addison-Wesley, Reading Mass. (1987). (File-use ratio)

Spivey 1989 J.M. Spivey, *The Z Notation: A Reference Manual,* Prentice Hall, London (1989).

Storer 1987 R. Storer, *Practical Program Development using JSP,* Blackwell Scientific, Oxford (1987).

Sutcliffe 1988 A. Sutcliffe, *Jackson System Development*, Prentice Hall, Hemel Hempstead (1988).

Taniar 1998 D. Taniar, "Towards an Ideal Placement Scheme for High-Performance Object-Oriented Database Systems", *Lecture Notes in Computer Science **1401**: High-Performance Computing and Networking: Proceedings HPCN Europe 1998, Amsterdam*, 508–517, (P. Sloot, M. Bubak & R. Hertzberger *eds.*), Springer (1998).

Taniar & Jiang 1998 D. Taniar and Y. Jiang, "A High-Performance Object-Oriented Distributed Parallel Database Architecture", *Lecture Notes in Computer Science **1401**: High-Performance Computing and Networking: Proceedings HPCN Europe 1998, Amsterdam*, 498–507, (P. Sloot, M. Bubak & R. Hertzberger *eds.*), Springer (1998).

Tarjan 1972, R.E. Tarjan, "Depth-first search and linear graph algorithms", *Society for Industrial and Applied Mathematics Journal of Computing*, **1**, 146–160, (1972).

Teichroew & Hershey 1977 D.Teichroew, and E.A. Hershey, "A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering*, **3**(1), 41–48 (Jan 1977).

Teichroew & Sayani 1971 D. Teichroew, and Sayani, H., "Automation of System Building", *Datamation*, **17**(8), 25–30, (Aug 1971).

Teisseire 1995 M. Teisseire, "Behavioural Constraints: Why using events instead of states", *Lecture Notes in Computer Science 1021: OOER '95: Object-oriented and Entity-Relationship modeling*, 123–132, (M.P. Papazoglou *ed.*), Springer (1995).

Thakur *et al.* 1996 R. Thakur, W. Gropp and E. Lusk, "An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application", *Lecture Notes in Computer Science* **1127**: *Parallel Computation: Proceedings 3rd International ACPC Conference, Klagenfurt*, 32–48, (L. Böszörményi *ed.*), Springer (1996).

Thinking Machines 1993 *CMMD Reference Manual, Version 3.0*, Thinking Machines Corp. (1993).

Tse & Pong 1986 T.H. Tse, and L. Pong, "An Application of Petri Nets in Structured Analysis", *ACM SIGSOFT Software Engineering Notes* **11**(5), 53–56 (Oct 1986).

Tse & Pong 1989 T.H. Tse, and L. Pong, "Towards a formal foundation for DeMarco data flow diagrams", *The Computer Journal* **32**(1), 1–12 (Feb 1989).

Veen 1986 Veen, A.H., *The Misconstrued Semicolon: Reconciling Imperative Languages and Dataflow Machines*, Centre for Mathematics and Computer Science, Amsterdam (1986).

Wang et al. 1997 J. Wang, J. Li and H. Kameda, "Scheduling Algorithms for Parallel Transaction Processing Systems", *Lecture Notes in Computer Science **1277**: Parallel Computing Technologies: Proceedings PaCT-97*, 283–297, (V. Malyshkin *ed.*), Springer (1997).

Ward & Mellor 1978 P.T. Ward & S.J. Mellor, *Structured Development for Real-Time Systems*, Yourdon Press, Englewood Cliffs, NJ. (1978).

Weinberg 1980 Weinberg, V., *Structured Analysis*, Prentice Hall, Englewood Cliffs, NJ (1980).

Weikum 1995 G. Weikum, "Tutorial on Parallel Database Systems", *Lecture Notes in Computer Science **893**: Database Theory: Proceedings ICDT '95*, 33–37, (G. Gottlob & M.Y. Vardi *eds.*), Springer (1995).

Weske 1998 M. Weske, "Object-Oriented design of a Flexible Workflow Management System", *Lecture Notes in Computer Science **1475**: Advances in Databases and Information Systems: Proceedings ADBIS '98*, 119–130, (W. Litwin, T. Morzy and G. Vossen *eds.*), Springer (1998).

Yourdon & Constantine 1978 Yourdon, E. and Constantine, L., *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Yourdon Press (1978).

13. Appendix: The Designer Program

Using Open-Prolog [Brady 1997] the *Designer* files may be consulted as follows:

```
:-[optimiser].
:-[parser].
:-[shared].
:-[analyser].
:-[canoniser].
:-[generator].
:-[designer].
```

The *Designer* program may then be invoked by a goal of the form:

```
        design(Source,Output).
```

where *Source* is the name of the specification file, and *Output* is the destination of the result.

The *Designer* program makes use of some system calls that are unique to Open-Prolog. These merely indicate the progress of the design phases, and summarise the times they take. The modules, as called from *Designer*, also make some system calls. To adapt the program to a different Prolog programming environment, these system calls should be modified. Alternatively, they may all be removed without ill effect.

```
design(Source,Output) :-
        'system$seconds'(T0),
        parse_spec(Source,Tree),
        'system$seconds'(T1),
        analyse_system(Tree,Decls,FinalDefs,Uses,Links),
        'system$seconds'(T2),
        canonise_graph(Uses,Links,SCCs,Root),
        'system$seconds'(T3),
        optimise_sccs(Decls,Uses,SCCs,Root,Processes,GraphOut),
        'system$seconds'(T4),
        generate_output(Output,Tree,Processes,GraphOut),
        'system$seconds'(T5),
        Parse is T1-T0,
        Analyse is T2-T1,
        Generate is T5-T4,
        Canonise is T3-T2,
        Optimise is T4-T3,
        length(Uses,N1),
        length(Links,N2),
        length(SCCs,N3),
        write('      Parse: '),write(Parse),write(' secs.'),nl,
        write('    Analyse: '),write(Analyse),write(' secs.'),nl,
        write('   Generate: '),write(Generate),write(' secs.'),nl,
        write('   Canonise: '),write(Canonise),write(' secs.'),nl,
        write('   Optimise: '),write(Optimise),write(' secs.'),nl,
        write('       Hard: '),write(N1),nl,
        write('       Soft: '),write(N2),nl,
        write('       SCCs: '),write(N3),nl.
```

13.1 The Parser

The *Parser* may be invoked directly by a goal of the form:

```
        parse(Source,Tree).
```

where *Source* is the specification file, and *Tree* is the file into which the abstract syntax tree should be written.

```
/* Parser.
This program parses a specification.
*/ parse(Source,Tree) :-
        parse_spec(Source,Spec),
        telling(User),tell(Tree),pretty(Spec),told,tell(User). /*
*/ parse_spec(Source,Spec) :-
        'system$push$display'(message,left,'Parsing specification...','','',''),
        scan_input(Source,Words),seen,see(user),specification(Spec,Words,Rest),
        ( Rest = []
        ; Rest=[Word|_] ->
            write(Word),write(' follows end of specification.'),nl,
            assert(error(_))
        ),
        'system$pop$display'(message),
        not clause(error(_),true),!. /*
*/ scan_input(Source,Words):-
            seeing(User),see(Source),get0(LookAhead),echo(LookAhead),
            skip_spaces(LookAhead,LookAhead1),
            read_all_words(LookAhead1,Words,_),!,seen,see(User). /*
*/ read_all_words(26,[],26) :- !. /*
*/ read_all_words(LookAhead,[Word|Words],LookAhead1) :-
                read_word(LookAhead,Word,LookAhead2),
                skip_spaces(LookAhead2,LookAhead3),
                read_all_words(LookAhead3,Words,LookAhead1). /*
*/ read_word(LookAhead,Word,LookAhead1):- valid_char(LookAhead),!,
            get0(LookAhead2),echo(LookAhead2),
            rest_name(LookAhead2,Rest,LookAhead1),
            lower_case(LookAhead,Lower),name(Word,[Lower|Rest]). /*
*/ read_word(LookAhead,Word,LookAhead1):- delimiter(LookAhead),!,
            get0(LookAhead1),echo(LookAhead1),name(Word,[LookAhead]). /*
*/ read_word(LookAhead,Op,LookAhead1):- valid_op(LookAhead),!,
            get0(LookAhead2),echo(LookAhead2),
            rest_operator(LookAhead2,Rest,LookAhead1),name(Op,[LookAhead|Rest]). /*
*/ rest_name(LookAhead,[Lower|Rest],LookAhead1):- valid_char(LookAhead),!,
            lower_case(LookAhead,Lower),
            get0(LookAhead2),echo(LookAhead2),
            rest_name(LookAhead2,Rest,LookAhead1). /*
*/ rest_name(LookAhead,[],LookAhead). /*
*/ rest_operator(LookAhead,[LookAhead|Rest],LookAhead1):- valid_op(LookAhead),!,
                get0(LookAhead2),echo(LookAhead2),
                rest_operator(LookAhead2,Rest,LookAhead1). /*
*/ rest_operator(LookAhead,[],LookAhead). /*
*/ skip_spaces(26,26) :- !. /*
*/ skip_spaces(LookAhead,LookAhead) :- delimiter(LookAhead),!. /*
*/ skip_spaces(LookAhead,LookAhead) :- valid_char(LookAhead),!. /*
*/ skip_spaces(LookAhead,LookAhead) :- valid_op(LookAhead),!. /*
*/ skip_spaces(LookAhead,LookAhead2) :-
            get0(LookAhead1),echo(LookAhead1),
            skip_spaces(LookAhead1,LookAhead2). /*
*/ digit(Char) :- Char>=48,Char=<57. /*
*/ letter(Char) :- Char>=65,Char=<90,!. /*
*/ letter(Char) :- Char>=97,Char=<122. /*
*/ lower_case(Char,Lower) :- Char>=65,Char=<90,!,Lower is Char+32. /*
*/ lower_case(Char,Char) :- true. /*
*/ valid_char(Char) :- letter(Char),!. /*
*/ valid_char(Char) :- digit(Char),!. /*
*/ valid_char(Char) :- name('_',[Char]),!. /*
*/ delimiter(Char)  :- name('(',[Char]),!. /*
*/ delimiter(Char)  :- name(')',[Char]),!. /*
*/ delimiter(Char)  :- name(';',[Char]),!. /*
*/ valid_op(Char) :- valid_char(Char),!,fail. /*
*/ valid_op(Char) :- delimiter(Char),!,fail. /*
*/ valid_op(Char) :- Char > 32. /*
*/ echo(_) :- !. /* ECHO IS SWITCHED OFF
*/ echo(26) :- !,write('(eof)'),nl. /*
*/ echo(31) :- !,nl. /*
*/ echo(9)  :- !,write('    '). /*
*/ echo(C)  :- C >= 32,!,put(C). /*
*/ echo(C)  :- write('\'),write(C). /*
*/ scan(Source,Listing) :-
      scan_input(Source,Words),tell(Listing),
      writeq(Words),write('.'),told,tell(user). /*
```

```
*/ specification(system(SysName,packages(Packages),
               generics(Generics),StateDecs,Events)) -->
       preamble(Packages,Generics),
       package_spec(SysName),
       package_body(SysName,StateDecs,Events). /*
*/ preamble(Packages,Generics) -->
               {abolish(error,1),abolish(function,1),abolish(package,1)},
               packages(Packages),generics(Generics),!. /*
*/ packages([Package|Packages]) --> package(Package),!,packages(Packages). /*
*/ packages([]) --> empty. /*
*/ package(Package) --> [with],identifier(Package),[';'],!,
                        {assert(package(Package))}. /*
*/ package(_) --> [with],!,syntax_past('package declaration',";"). /*
*/ generics(Generics) --> [generic],!,generic_types(Generics). /*
*/ generics([]) --> empty. /*
*/ generic_types([TypeName|TypeNames]) --> generic_type(TypeName),!,
                                           generic_types(TypeNames). /*
*/ generic_types([]) --> empty. /*
*/ generic_type(Type) --> [type],identifier(Type),[is],[private],[';'],!. /*
*/ generic_type(Type) --> [type],identifier(Type),[is],[range],['<>'],[';'],!. /*
*/ generic_type(_)    --> [type],!,syntax_past('generic type declaration',';'). /*
*/ package_spec(SysName) -->
               {abolish(current_num,2)},
               [package],identifier(SysName),[is],
                event_specs,
               [end],identifier(SysName2),[';'],
               {consistent(SysName,SysName2,'end of package specification')},!. /*
*/ package_spec(SysName) -->
               {abolish(current_num,2)},
               [package],identifier(SysName),[is],event_specs,!,
               syntax_to('event declarations','package'). /*
*/ package_spec(SysName) -->
               {abolish(current_num,2)},
               syntax_to('package specification','package'). /*
*/ package_body(SysName,StateDecs,Events) -->
               {abolish(current_num,2)},
               [package],[body],identifier(SysName2),[is],
               state_variables(SysName2,[],StateDecs,[],Globals),
               functions,
               events(Events,Globals),
               [end],identifier(SysName3),[';'],!,
               {consistent(SysName,SysName2,'package body')},
               {consistent(SysName,SysName3,'end of package body')},!. /*
*/ package_body(SysName,StateDecs,Events) -->
               {abolish(current_num,2)},
               [package],[body],identifier(SysName2),[is],
               state_variables(SysName2,[],StateDecs,[],Globals),
               functions,
               events(Events,Globals),!,
               syntax_to('event implementations',SysName). /*
*/ package_body(SysName,StateDecs,Events) -->
               {abolish(current_num,2)},
               [package],[body],identifier(SysName2),[is],
               state_variables(SysName2,[],StateDecs,[],Globals),
               functions,!,
               syntax_to('event implementations',SysName). /*
*/ package_body(SysName,StateDecs,Events) -->
               {abolish(current_num,2)},
               [package],[body],identifier(SysName2),[is],
               state_variables(SysName2,[],StateDecs,[],Globals),!,
               syntax_to('function implementations',SysName). /*
*/ package_body(SysName,StateDecs,Events) -->
               {abolish(current_num,2)},
               [package],[body],identifier(SysName2),[is],!,
               syntax_to('state variable declarations',SysName). /*
*/ package_body(SysName,StateDecs,Events) -->
               {abolish(current_num,2)},
               [package],!,
               syntax_to('package body heading',SysName). /*
*/ state_variables(SysName,DecsIn,DecsOut,SymsIn,SymsOut) -->
                state_var_dec(SysName,DecsIn,Decs1,SymsIn,Syms1),!,
                state_variables(SysName,Decs1,DecsOut,Syms1,SymsOut). /*
*/ state_variables(_,Decs,Decs,Syms,Syms) --> empty. /*
*/ parameters(Event,DecsIn,DecsOut,SymsIn,SymsOut) -->
             parameter_dec(Event,DecsIn,Decs1,SymsIn,Syms1),!,
             parameter_tail(Event,Decs1,DecsOut,Syms1,SymsOut). /*
*/ parameter_tail(Event,DecsIn,DecsOut,SymsIn,SymsOut) --> [';'],
               parameters(Event,DecsIn,DecsOut,SymsIn,SymsOut),!. /*
*/ parameter_tail(Event,DecsIn,DecsOut,SymsIn,SymsOut) --> [';'],!,
               syntax_past('parameter list',';'). /*
*/ parameter_tail(_,Decs,Decs,Syms,Syms) --> empty. /*
*/ local_variables(Event,DecsIn,DecsOut,SymsIn,SymsOut) -->
                local_var_dec(Event,DecsIn,Decs1,SymsIn,Syms1),!,
                local_variables(Event,Decs1,DecsOut,Syms1,SymsOut). /*
*/ local_variables(_,Decs,Decs,Syms,Syms) --> empty. /*
*/ state_var_dec(SysName,DecsIn,DecsOut,SymsIn,SymsOut) -->
              id_list(Ids),[':'],type_dec(Dom,Codom),initialiser,[';'],!,
          {define_state_vars(SysName,Ids,Dom,Codom,DecsIn,DecsOut,SymsIn,SymsOut)}. /*
*/ state_var_dec(_,Decs,Decs,Syms,Syms) -->
              id_list(_),[':'],!,syntax_past('state declaration',';'). /*
*/ local_var_dec(Event,DecsIn,DecsOut,SymsIn,SymsOut) -->
              id_list(Ids),[':'],type_dec(Dom,Codom),initialiser,[';'],!,
           {define_local_vars(Event,Ids,Dom,Codom,DecsIn,DecsOut,SymsIn,SymsOut)}. /*
*/ local_var_dec(_,Decs,Decs,Syms,Syms) -->
              id_list(_),[':'],!,syntax_past('local declaration',';'). /*
*/ parameter_dec(Event,DecsIn,DecsOut,SymsIn,SymsOut) -->
              id_list(Ids),[':'],type_dec(Dom,Codom),!,
           {define_parameters(Event,Ids,Dom,Codom,DecsIn,DecsOut,SymsIn,SymsOut)}. /*
*/ parameter_dec(_,Decs,Decs,Syms,Syms) -->
              id_list(_),[':'],!,syntax_past('parameter declaration',';'). /*
*/ type_dec(Dom,Codom)-->[array],['('],id_list(Dom),[')'],[of],identifier(Codom),!. /*
*/ type_dec(_,_) --> [array],!,syntax_to('array type declaration',';'). /*
*/ type_dec([],Codom) --> identifier(Codom),!. /*
*/ type_dec(_,_) --> syntax_to('simple type declaration',';'). /*
*/ id_list([Id|Ids]) --> identifier(Id),!,id_tail(Ids). /*
*/ id_list([]) --> empty. /*
*/ id_tail(Ids) --> [','],id_list(Ids),!. /*
*/ id_tail(Ids) --> [','],!,syntax_past('identifier list',';'). /*
*/ id_tail([]) --> empty. /*
*/ event_specs --> event_spec,!,event_spec_tail. /*
*/ event_spec_tail --> event_specs. /*
*/ event_spec_tail --> empty. /*
*/ event_spec --> [procedure],identifier(Event),
                  ['('],parameters(Event,[],_,[],_),[')'],[';'],!. /*
*/ event_spec --> [procedure],identifier(Event),[';'],!. /*
*/ event_spec --> [procedure],!,syntax_past('event specification',';'). /*
*/ events([Event|Events],Globals) -->
       event(Event,Globals),!,
       event_tail(Events,Globals). /*
*/ event_tail(Events,Globals) --> events(Events,Globals). /*
*/ event_tail([],_) --> empty. /*
*/ event(event(Event,ParamDecs,LocalDecs,Body),Globals) -->
      [procedure],identifier(Event),
      ['('],parameters(Event,[],ParamDecs,Globals,Params),[')'],[is],
      local_variables(Event,[],LocalDecs,Params,Syms),
      [begin],
        statements(Event,Syms,_,Body),
      [end],identifier(Event2),[';'],!,
      {consistent(Event,Event2,'end of event body')}. /*
*/ event(event(Event,[],LocalDecs,Body),Globals) -->
      [procedure],identifier(Event),[is],
      local_variables(Event,[],LocalDecs,Globals,Syms),
      [begin],
        statements(Event,Syms,_,Body),
      [end],identifier(Event2),[';'],!,
      {consistent(Event,Event2,'end of event body')}. /*
*/ event(null,_) -->
      [procedure],identifier(Event),
      ['('],parameters(Event,[],_,[],_),[')'],[is],
      local_variables(Event,[],_,[],_),
      [begin],!,syntax_past('event body','end'),skip_past(';'). /*
*/ event(null,_) -->
      [procedure],!,syntax_past('event heading','end'),skip_past(';'). /*
*/ functions --> function,!,functions. /*
*/ functions --> empty. /*
*/ function -->
          [function],identifier(FunctionName),{assert(function(FunctionName))},
          ['('],parameters(FunctionName,[],Decs1,[],Syms1),[')'],
          [return],identifier(_),[is],
          local_variables(FunctionName,Decs1,Decs2,Syms1,Syms2),
          [begin],
             statements(FunctionName,Syms2,_,_),
          [end],identifier(_),[';'],!. /*
*/ function(_) --> [function],!,
          syntax_past('function declaration','end'),skip_past(';'). /*
*/ statements(Event,Syms,SymsOut,[Stmt|Stmts]) -->
            statement(Event,Syms,Syms1,Stmt),!,
            statement_tail(Event,Syms1,SymsOut,Stmts). /*
*/ statement_tail(Event,Syms,SymsOut,Stmts) -->
            statements(Event,Syms,SymsOut,Stmts),!. /*
*/ statement_tail(_,Syms,Syms,[]) --> empty. /*
The predicate 'statement(+Event,+SymsIn,-SymsOut,-SyntaxTree)' parses a statement producing a syntax tree.
Because some statements may declare new variables, SymsOut may be a superset of SymsIn.
*/ statement(_,Syms,Syms,null) --> [null],[;],!. /*
*/ statement(_,Syms,Syms,return(Expn)) --> [return],[;],!. /*
*/ statement(_,Syms,Syms,return(Expn)) --> [return],expression(Syms,Expn),[;],!. /*
*/ statement(_,Syms,Syms,_) --> [return],syntax_past('return statement',';'). /*
*/ statement(Event,Syms,SymsOut,if(lex([internal/Event/Var],1),Expn,True,False)) -->
           [if],expression(Syms,Expn),{exp_name(Event,Var)},[then],
             statements(Event,Syms,Syms1,True),
            elsif_part(Event,Syms,Syms2,False),
            {append(Syms1,Syms2,Syms3),sort(Syms3,SymsOut)},
           [end],[if],[;],!. /*
*/ statement(_,Syms,Syms,_) -->
           [if],syntax_past('if statement','end'),skip_past(;). /*
*/ elsif_part(Event,Syms,SymsOut,[if(lex([internal/Event/Var],1),Expn,True,False)])
        --> [elsif],!,expression(Syms,Expn),{exp_name(Event,Var)},[then],
             statements(Event,Syms,Syms1,True),
            elsif_part(Event,Syms,Syms2,False),
            {append(Syms1,Syms2,Syms3),sort(Syms3,SymsOut)},!. /*
*/ elsif_part(Event,Syms,SymsOut,Stmts) -->
            [else],statements(Event,Syms,SymsOut,Stmts),!. /*
*/ elsif_part(_,Syms,Syms,_) -->
            [elsif],!,syntax_past('elsif clause','end'),skip_past(;). /*
*/ elsif_part(_,Syms,Syms,_) -->
            [else],!,syntax_past('else clause','end'),skip_past(;). /*
*/ elsif_part(_,Syms,Syms,[]) --> empty. /*
*/ statement(Event,Syms,SymsOut,while(lex([internal/Event/Var],1),Expn,Body)) -->
           [while],expression(Syms,Expn),{exp_name(Event,Var)},[loop],
             statements(Event,Syms,SymsOut,Body),
           [end],[loop],[;],!. /*
*/ statement(_,Syms,Syms,_) -->
           [while],syntax_past('while statement','end'),skip_past(;). /*
*/ statement(Event,Syms,SymsOut,all(lex([loop/Event/Var],N),Codom,Body)) -->
           [all],identifier(Var),
           {local_check(Event,Var,Syms),
            Sym=(loop/Event/Var,[],Codom),
            unique(loop/Event/Var,N)},
           [in],identifier(Codom),[loop],
             statements(Event,[Sym|Syms],SymsOut,Body),
           [end],[loop],[;],!. /*
*/ statement(_,Syms,Syms,_) -->
           [all],syntax_past('all statement','end'),skip_past(;). /*
*/ statement(Event,Syms,SymsOut,for(lex([loop/Event/Var],N),Codom,Body)) -->
           [for],identifier(Var),
           {local_check(Event,Var,Syms),
            Sym=(loop/Event/Var,[],Codom),
            unique(loop/Event/Var,N)},
           [in],identifier(Codom),[loop],
             statements(Event,[Sym|Syms],SymsOut,Body),
           [end],[loop],[;],!. /*
*/ statement(_,Syms,Syms,_) -->
           [for],syntax_past('for statement','end'),skip_past(;). /*
```

```
*/ statement(Event,Syms,SymsOut,declare(LocalDecs,Body)) -->
            [declare],
              local_variables(Event,[],LocalDecs,Syms,Syms1),
            [begin],
              statements(Event,Syms1,SymsOut,Body),
            [end],[';'],!. /*
*/ statement(_,Syms,Syms,[]) -->
            [declare],!,syntax_past('declare block','end'),skip_past(';'). /*
*/ statement(_,Syms,Syms,call(lex([output/Package/Event],N),Expn_List)) -->
            identifier(Package),['.'],identifier(Event),
            {unique(output/Package/Event,N)},
            ['('],exp_list(Syms,Expn_List),[')',';'],!. /*
*/ statement(_,Syms,Syms,call(lex([output/Package/Event],N),[])) -->
            identifier(Package),['.'],identifier(Event),[';'],!,
            {unique(output/Package/Event,N)}. /*
*/ statement(_,Syms,Syms,_) -->
            identifier(_),['.'],identifier(_),
            ['('],!,syntax_past('delayed call parameters',');'),!. /*
*/ statement(_,Syms,Syms,_) -->
            identifier(_),['.'],!,syntax_past('delayed call',');'),!. /*
*/ statement(_,Syms,Syms,assign(Var,Expn)) -->
            variable(Syms,Var),[:=],expression(Syms,Expn),[;],!. /*
*/ statement(_,Syms,Syms,_) -->
            variable(Syms,_),[:=],!,syntax_past('assignment',';'). /*
*/ variable(Syms,lex([Symbol|Subscripts],N)) -->
           identifier(Id),
           {lookup(Id,Syms,(Symbol,Dom,Codom)),
           unique(Symbol,N)},
           subscript_list(Id,Dom,Syms,Subscripts). /*
*/ subscript_list(Id,Dom,Syms,Vars) --> ['('],var_list(Id,Dom,Syms,Vars),[')'],!. /*
*/ subscript_list(_,_,_,_) --> ['('],!,syntax_past('subscript list',')'). /*
*/ subscript_list(_,[],_,[]) --> empty. /*
*/ subscript_list(Array,[_|_],_,[]) -->
                 {write('"'),write(Array),write('" should be indexed.'),nl}. /*
*/ var_list(Array,[Dom|Doms],Syms,[Var|Vars]) --> identifier(Id),!,
           {lookup(Id,Syms,(Var,Simple,Codom)),
            (Simple==[], Codom==Dom, local_var(Var);
             write('"'),write(Id),write('" has wrong type to index "'),
             write(Array),write('"'),nl
            )
           },
           var_tail(Array,Doms,Syms,Vars). /*
*/ var_list(Array,[_|_],Syms,[]) --> !,
           {write('Not enough subscripts of "'),write(Array),write('"'),nl}. /*
*/ var_list(_,[],_,[]) --> empty. /*
*/ var_list(Array,[],Syms,[]) --> identifier(Id),!,
           {write('"'),write(Id),write('" is an excess subscript of "'),
            writeq(Array),write('"'),nl},
           var_tail(Array,[],Syms,_). /*
*/ var_tail(Array,Dom,Syms,Vars) --> [','],var_list(Array,Dom,Syms,Vars),!. /*
*/ var_tail(Array,Dom,Syms,Vars) --> [','],!,syntax_after('variable list',';'). /*
*/ var_tail(_,[],_,[]) --> !. /*
*/ var_tail(Array,[_|_],_,[]) --> {write('Missing subscript(s) for "'),
                                  write(Array),write('"'),nl}. /*
*/ local_var(local/_/_) :- !. /*
*/ local_var(loop/_/_) :- !. /*
*/ local_var(input/_/_) :- !. /*
*/ local_var(internal/_/_) :- !. /*
*/ expression(Syms,Vars) -->
            term(Syms,Vars1),
            more_terms(Syms,Vars2),
            {append(Vars1,Vars2,Vars)}. /*
*/ term(Syms,Vars)   --> prefix_operator,!,term(Syms,Vars). /*
*/ term(Syms,Vars)   --> ['('],expression(Syms,Vars),[')'],!. /*
*/ term(_,_)         --> ['('],!,syntax_past('nested expression',')'). /*
*/ term(_,[])        --> constant,!. /*
*/ term(Syms,Expns) --> identifier(FunctionName),
                        {clause(function(FunctionName),true)},
                        ['('],exp_list(Syms,Expns),[')'],!. /*
*/ term(_,_)         --> identifier(FunctionName),
                        {clause(function(FunctionName),true)},!,
                        syntax_past('function call',')'). /*
*/ term(Syms,[Var]) --> variable(Syms,Var),!. /*
*/ term(_,_)         --> syntax_to('term',';'). /*
*/ more_terms(Syms,Vars) --> infix_operator,expression(Syms,Vars),!. /*
*/ more_terms(Syms,Vars) --> infix_operator,!,syntax_past('infix operator',';'). /*
*/ more_terms(_,[]) --> empty. /*
*/ exp_list(Syms,Vars) -->
          expression(Syms,Vars1),
          exp_tail(Syms,Vars2),{append(Vars1,Vars2,Vars)}. /*
*/ exp_tail(Syms,Vars) --> [','],exp_list(Syms,Vars),!. /*
*/ exp_tail(Syms,Vars) --> [','],!,syntax_past('expression list',';'). /*
*/ exp_tail(_,[]) --> empty. /*
*/ initialiser --> [:=],constant,!. /*
*/ initialiser --> [:=],mapping,!. /*
*/ initialiser --> [:=],!,syntax_to('initialiser',';'). /*
*/ initialiser --> empty. /*
*/ mapping --> ['('],[others],[=>],mapping,[')'],!. /*
*/ mapping --> ['('],[others],[=>],constant,[')']. /*
*/ infix_operator --> [+]. /*
*/ infix_operator --> [-]. /*
*/ infix_operator --> [*]. /*
*/ infix_operator --> [/]. /*
*/ infix_operator --> [mod]. /*
*/ infix_operator --> [&]. /*
*/ infix_operator --> [or]. /*
*/ infix_operator --> [and]. /*
*/ infix_operator --> [=]. /*
*/ infix_operator --> [/=]. /*
*/ infix_operator --> [<=]. /*
*/ infix_operator --> [<]. /*
*/ infix_operator --> [=>]. /*
*/ infix_operator --> [>]. /*
*/ prefix_operator --> [+]. /*
*/ prefix_operator --> [-]. /*
*/ prefix_operator --> [not]. /*
*/ empty(S,S). /*
*/ reserved_word(and). /*
*/ reserved_word(array). /*
*/ reserved_word(begin). /*
*/ reserved_word(body). /*
*/ reserved_word(else). /*
*/ reserved_word(elsif). /*
*/ reserved_word(end). /*
*/ reserved_word(function). /*
*/ reserved_word(generic). /*
*/ reserved_word(if). /*
*/ reserved_word(in). /*
*/ reserved_word(is). /*
*/ reserved_word(mod). /*
*/ reserved_word(not). /*
*/ reserved_word(of). /*
*/ reserved_word(or). /*
*/ reserved_word(out). /*
*/ reserved_word(not). /*
*/ reserved_word(package). /*
*/ reserved_word(private). /*
*/ reserved_word(procedure). /*
*/ reserved_word(return). /*
*/ reserved_word(subtype). /*
*/ reserved_word(then). /*
*/ reserved_word(type). /*
*/ reserved_word(with). /*
*/ identifier(Id,[Id|S],S) :- reserved_word(Id),!,fail. /*
*/ identifier(Id,[Id|S],S) :- name(Id,[C|Rest]),letter(C). /*
*/ define_state_vars(SysName,[Id|Ids],Dom,Codom,DecsIn,DecsOut,SymsIn,SymsOut):-
               state_check(SysName,Id,SymsIn),
               Name=global/SysName/Id,Sym=(Name,Dom,Codom),unique(Name,N),
               Decs1 = [state(lex([Name|Dom],N),Codom)|DecsIn],
               Syms1 = [Sym|SymsIn],
               define_state_vars(SysName,Ids,Dom,Codom,Decs1,DecsOut,Syms1,SymsOut). /*
*/ define_state_vars(_,[],_,_,Decs,Decs,Syms,Syms). /*
*/ define_local_vars(Event,[Id|Ids],Dom,Codom,DecsIn,DecsOut,SymsIn,SymsOut):-
                local_check(Event,Id,SymsIn),
                Name=local/Event/Id,Sym=(Name,Dom,Codom),unique(Name,N),
                Decs1 = [local(lex([Name|Dom],N),Codom)|DecsIn],
                Syms1 = [Sym|SymsIn],
                define_local_vars(Event,Ids,Dom,Codom,Decs1,DecsOut,Syms1,SymsOut). /*
*/ define_local_vars(_,[],_,_,Decs,Decs,Syms,Syms). /*
*/ define_parameters(Event, [Id|Ids],Dom,Codom,DecsIn,DecsOut,SymsIn,SymsOut) :-
                local_check(Event,Id,SymsIn),
                Name=input/Event/Id,Sym=(Name,Dom,Codom),unique(Name,N),
                Decs1 = [param(lex([Name|Dom],N),Codom)|DecsIn],
                Syms1 = [Sym|SymsIn],
                define_parameters(Event,Ids,Dom,Codom,Decs1,DecsOut,Syms1,SymsOut). /*
*/ define_parameters(_,[],_,_,Decs,Decs,Syms,Syms). /*
*/ lookup(Id,Syms,Var) :- find_id(loop/_/Id,Syms,Var),!. /*
*/ lookup(Id,Syms,Var) :- find_id(local/_/Id,Syms,Var),!. /*
*/ lookup(Id,Syms,Var) :- find_id(input/_/Id,Syms,Var),!. /*
*/ lookup(Id,Syms,Var) :- find_id(_/_/Id,Syms,Var),!. /*
*/ lookup(Id,Syms,('?'/'?'/Id,[],[])) :-
        writeq(Id),write(' is not declared.'),nl,pretty(Syms). /*
*/ local_check(Event,Id,Syms) :-
            find_id(local/_/Id,Syms,_),!,writeq(Id),
            write(' is already declared as a local variable in '),writeq(Event),nl. /*
*/ local_check(Event,Id,Syms) :-
            find_id(loop/_/Id,Syms,_),!,writeq(Id),
            write(' is already declared as a loop variable in '),writeq(Event),nl. /*
*/ local_check(Event,Id,Syms) :-
            find_id(input/_/Id,Syms,_),!,writeq(Id),
            write(' is already declared as a parameter in '),writeq(Event),nl. /*
*/ local_check(_,_,_) :- true. /*
*/ state_check(System,Id,Syms) :-
         find_id(global/_/Id,Syms,_),!,writeq(Id),
          write(' is already declared as a state variable in '),writeq(System),nl. /*
*/ state_check(_,_,_) :- true. /*
*/ find_id(Id,[(Id,Dom,Codom)|Syms],(Id,Dom,Codom)) :- !. /*
*/ find_id(Id,[_|Syms],Symbol) :- find_id(Id,Syms,Symbol). /*
*/ constant --> [true]. /*
*/ constant --> [false]. /*
*/ constant --> [null]. /*
*/ constant([N|S],S) :- name(N,[C|Rest]),digit(C). /*
*/ syntax_past(Msg,Stop,[H|T],U) :-
             write('Syntax error in '),write(Msg),write(' at or after "'),
             write(H),write('".'),nl,
             skip_past(Stop,[H|T],U),assert(error(syntax)). /*
*/ syntax_to(Msg,Stop,[H|T],U) :-
           write('Syntax error in '),write(Msg),write(' at or after "'),
           write(H),write('".'),nl,
           skip_to(Stop,[H|T],U),assert(error(syntax)). /*
*/ skip_past(X,[X|S],S) :- !. /*
*/ skip_past(X,[_|S1],S2) :- skip_past(X,S1,S2). /*
*/ skip_to(X,[X|S],[X|S]) :- !. /*
*/ skip_to(X,[_|S1],S2) :- skip_to(X,S1,S2). /*
*/ consistent(P1,P1,Msg):- !. /*
*/ consistent(P1,P2,Msg):-
            write(Msg),write(' should be named "'),
            write(P1),write('" not "'),write(P2),write('".'),nl. /*
 */ exp_name(Event,Var) :-
            unique('internal/Event/expn_',N),
            name(N,Digits),
            append("expn_",Digits,Name),
            name(Var,Name). /*
End of parser. */
```

13.2 Shared Predicates used by other Modules

The *Shared* module contains predicates that are called by all the modules below. It is not intended to be invoked directly.

```
/* Shared predicates.
The following predicates are modified from the SICStus Prolog library.
A graph is usually represented as an ordered list of (Vertex-Successors) pairs, where Successors is an ordered
list of the vertices that are the successors of Vertex in the graph, i.e., there is an edge from Vertex to each
member of Successors.
The predicate 'vertices_edges_to_graph(+Vertices,+Edges,-Graph)' succeeds if Vertices is a list of vertices, Edges
is a list of edges, and Graph is a graph built from Vertices and Edges. Vertices and Edges may be in any order.
The vertices mentioned in Edges do not have to occur explicitly in Vertices. Vertices may be used to specify
vertices that are not connected to any edges.
```

```
 */ vertices_edges_to_graph(Vertices0,Edges,Graph) :-
        sort(Vertices0,Vertices1),sort(Edges,EdgeSet),
        edges_vertices(EdgeSet,Bag),sort(Bag,Vertices2),
        ord_union(Vertices1,Vertices2,VertexSet),
        group_edges(VertexSet,EdgeSet,Graph). /*
 */ edges_vertices([],[]). /*
 */ edges_vertices([From-To|Edges],[From,To|Vertices]) :-
      edges_vertices(Edges,Vertices). /*
 */ group_edges([],_,[]). /*
 */ group_edges([Vertex|Vertices],Edges,[Vertex-Successors|G]) :-
      group_edges(Edges,Vertex,Successors,RestEdges),
      group_edges(Vertices,RestEdges,G). /*
 */ group_edges([V0-X|Edges],V,[X|Successors],RestEdges) :- V0==V,!,
      group_edges(Edges,V,Successors,RestEdges). /*
 */ group_edges(Edges,_,[],Edges). /*
```
The predicate 'vertices(+Graph,-Vertices)' unifies Vertices with the vertices in Graph.
```
 */ vertices([],[]). /*
 */ vertices([Vertex-_|Graph],[Vertex|Vertices]) :- vertices(Graph,Vertices). /*
```
The predicate 'edges(+Graph,-Edges)' unifies Edges with the edges in Graph.
```
 */ edges([],[]). /*
 */ edges([Vertex-Successors|G],Edges) :-
     vertex_edges(Successors,Vertex,Edges,MoreEdges),
     edges(G,MoreEdges). /*
 */ vertex_edges([],_,Edges,Edges). /*
 */ vertex_edges([Successor|Successors],Vertex,[Vertex-Successor|Edges],MoreEdges) :-
      vertex_edges(Successors,Vertex,Edges,MoreEdges). /*
```
The predicate 'transpose(+Graph,-Transpose)' is true if Transpose is the graph computed by replacing each edge (u,v) in Graph by its symmetric edge (v,u). It can only be used one way around. The cost is O(N log N).
```
 */ transpose(Graph,Transpose) :-
          transpose_edges(Graph,TEdges,[]),sort(TEdges,TEdges2),
          vertices(Graph,Vertices),group_edges(Vertices,TEdges2,Transpose). /*
 */ transpose_edges([],Edges,Edges) :- !. /*
 */ transpose_edges([Vertex-Successors|G],Edges1,Edges3) :-
          transpose_edges(Successors,Vertex,Edges1,Edges2),
          transpose_edges(G,Edges2,Edges3). /*
 */ transpose_edges([],_,Edges,Edges) :- !. /*
 */ transpose_edges([Successor|Successors],Vertex,Edges1,Edges3) :-
          Edges1=[Successor-Vertex|Edges2],
          transpose_edges(Successors,Vertex,Edges2,Edges3). /*
```
The predicate 'neighbours(+Vertex,+Graph,-Successors)' is true if Vertex is a vertex in Graph and Successors are its neighbours.
```
 */ neighbours(V,[V0-Successors|_],Successors) :- V0==V,!. /*
 */ neighbours(V,[_|Graph],Successors) :- neighbours(V,Graph,Successors). /*
```
The predicate 'transitive_closure(+Graph,-Closure)' computes Closure as the transitive closure of Graph in O(N^3) time.
```
 */ transitive_closure(Graph,Closure) :- !,warshall(Graph,Graph,Closure). /*
 */ warshall([],Closure,Closure). /*
 */ warshall([V-_|Vs],Graph,Closure) :-
        neighbours(V,Graph,Y),
        warshall_vertex(Graph,V,Y,NewGraph),
        warshall(Vs,NewGraph,Closure). /*
 */ warshall_vertex([],_,_,[]). /*
 */ warshall_vertex([X-Successors|G],V,Y,[X-NewSuccessors|NewG]) :-
        ord_subset([V],Successors),!,
        ord_del_element(Y,X,Y1),
        ord_union(Successors,Y1,NewSuccessors),
        warshall_vertex(G,V,Y,NewG). /*
 */ warshall_vertex([X-Successors|G],V,Y,[X-Successors|NewG]) :-
        warshall_vertex(G,V,Y,NewG). /*
```
The predicate 'subgraph(+Graph,+Vertices,-Subgraph)' succeeds when Subgraph is a graph whose vertices are common to Vertices and Graph, and whose edges are the subset of those in Graph that join the vertices in Subgraph.
```
 */ subgraph([],_,[]). /*
 */ subgraph([V-Successors|Graph],Vs,[V-Successors1|Subgraph]) :-
        ord_subset([V],Vs),!,
        ord_intersection(Successors,Vs,Successors1),
        subgraph(Graph,Vs,Subgraph). /*
 */ subgraph([_|Graph],Vs,Subgraph) :- subgraph(Graph,Vs,Subgraph). /*
```
The predicate 'reduce(+Graph,-Reduced)' is true if Reduced is the reduced graph for Graph. The vertices of the reduced graph are the strongly connected components of Graph. There is an edge in Reduced from u to v iff there is an edge in Graph from one of the vertices in u to one of the vertices in v. A strongly connected component is a maximal set of vertices where each vertex has a path to every other vertex. Algorithm from "Algorithms" by Sedgewick,page 482, Tarjan's algorithm. Approximately linear in the maximum of arcs and nodes (O(N log N)).

```
 */ reduce(Graph,Reduced,SCCs) :-
      strong_components(Graph,SCCs,Map),
      reduced_vertices_edges(Graph,Vertices,Map,Edges,[]),
      sort(Vertices,Vertices1),sort(Edges,Edges1),
      group_edges(Vertices1,Edges1,Reduced),sort(SCCs,Vertices1). /*
```

The predicate 'strong_components(Graph,SCCs,Map)' succeeds iff SCCs are the strongly connected components of graph Graph, and Map associates each component in SCCs with a set of vertices in Graph.

```
 */ strong_components(Graph,SCCs,A) :-
       nodeinfo(Graph,Nodeinfo,Vertices),
       ord_list_to_assoc(Nodeinfo,A0),
       visit(Vertices,0,_,A0,A,0,_,[],_,SCCs,[]). /*
```

The predicate 'visit(+Vertices,Min1,Min2,Info,Map,PreorderNum,Stack1,Stack2, SCCs1,SCCs2)' implements a depth-first traversal of a graph whose vertices are Vertices, and whose edges are represented by the list 'Info', which associates a triple of the form 'node(Successors,MinSucc,_)' with each vertex.

Each vertex is considered in turn. The call of 'get_assoc' finds the information for vertex V. A visited vertex has a MinSucc>0, which is not updated.

An unvisited vertex has a MinSucc of 0, and results its MinSucc being updated to its PreorderNum+1, followed by a recursive call of visit to all its successors, with PreorderNum increased by 1.

```
 */ visit([],Min,Min,A,A,I,I,Stk,Stk,SCCs,SCCs) :- !. /*
 */ visit([V|Vs],Min0,Min,A0,A,I,M,Stk0,Stk,SCCs1,SCCs4) :-
     get_assoc(V,A0,node(Ns,J,Eq),A1,node(Ns,K,Eq)),
     (J>0 -> J=K,J=Min1,A1=A3,I=L,Stk0=Stk2,SCCs3=SCCs1;
      K is I+1,visit(Ns,K,Min1,A1,A2,K,L,[V|Stk0],Stk1,SCCs1,SCCs2),
      (K>Min1->A2=A3,Stk1=Stk2,SCCs3=SCCs2;
       pop(V,Eq,A2,A3,Stk1,Stk2,[],SCCs2,SCCs3)
      )
          ),
          (Min0<Min1 -> Min2=Min0; Min2=Min1),
    visit(Vs,Min2,Min,A3,A,L,M,Stk2,Stk,SCCs3,SCCs4). /*
```

The predicate 'pop(Vertex,Eq,Info1,Info2,Stack1,Stack2,SCC0,-SCCs1,+SCCs2)' tests if Vertex is the top of the stack Stack1. If it is, it represents the first vertex visited of a strongly connected component, and SCC0 contains the remaining vertices of the component. It pops Stack1 giving Stack2, adds the top of stack to the list SCC0, sorts the updated SCC0, and adds it to the list of strong components SCCs2 giving SCCs1. If Vertex is not the top of stack Stack1, it adds the top of stack to the list SCC0, and calls itself recursively.

In either case, the MinSucc associated with the vertex is set to a high value, to prevent it being misused by a later visit.

```
 */ pop(V,Eq,A0,A, [V1|Stk0],Stk,SCC0,SCCs1,SCCs2) :-
       get_assoc(V1,A0,node(Ns,_,Eq),A1,node(Ns,9999999,Eq)),
       (V==V1 -> SCCs1=[SCC|SCCs2],A1=A,Stk0=Stk,sort([V1|SCC0],SCC);
        pop(V,Eq,A1,A,Stk0,Stk, [V1|SCC0],SCCs1,SCCs2)
       ). /*
```

The predicate 'node_info(+Graph,-NodeList,-Vertices)' succeeds iff graph 'Graph' has vertices 'Vertices' and 'NodeList' is a corresponding list of vertices, each paired with a triple of the form 'node(Successors,0,_)'.

```
 */ nodeinfo([],[],[]). /*
 */ nodeinfo([V-Ns|G],[V-node(Ns,0,_)|Nodeinfo],[V|Vs]) :- nodeinfo(G,Nodeinfo,Vs). /*
```

The predicate 'reduced_vertices_edges(+Graph,-Vertices,+Map,-EdgesOut,+EdgesIn)' places the vertices of the reduced graph of Graph into Vertices, and adds its edges to EdgesIn giving EdgesOut, with the aid of Map, which maps each SCC to a set of vertices.

```
 */ reduced_vertices_edges([],[],_,Edges,Edges) :- !. /*
 */ reduced_vertices_edges([V-Successors|Graph],[V1|Vs],Map,Edges1,Edges3) :-
      get_assoc(V,Map,N),N=node(_,_,V1),
      reduced_edges(Successors,V1,Map,Edges1,Edges2),
      reduced_vertices_edges(Graph,Vs,Map,Edges2,Edges3). /*
 */ reduced_edges([],_,_,Edges,Edges) :- !. /*
 */ reduced_edges([V|Vs],V1,Map,Edges1,Edges3) :-
       get_assoc(V,Map,N),N=node(_,_,V2),
       (V1==V2->Edges2=Edges1; Edges1=[V1-V2|Edges2]),
       reduced_edges(Vs,V1,Map,Edges2,Edges3). /*
```

The predicate 'reachable(+Vertex,+Graph,-Reachable)' is given a Graph and a Vertex of that Graph,and returns the set of vertices that are Reachable from that Vertex. Takes O(N^2) time.

```
 */ reachable(Initial,Graph,Reachable) :-
        reachable_vertex([Initial],Graph,[Initial],Reachable). /*
 */ reachable_vertex([],_,Reachable,Reachable). /*
 */ reachable_vertex([Q|R],Graph,Reach0,Reachable) :-
        neighbours(Q,Graph,Successors),!,
        ord_union(Reach0,Successors,Reach1,New),
        append(R,New,S),
        reachable_vertex(S,Graph,Reach1,Reachable). /*
 */ reachable_vertex([Q|R],Graph,Reach0,Reachable) :-
        reachable_vertex(R,Graph,Reach0,Reachable). /*
```

The predicate 'merge_graphs(+L,?G)' succeeds when G is the graph comprising the subgraphs in the list L, without duplicates.

```
 */ merge_graphs([],gr([],[])). /*
 */ merge_graphs([gr(V,E)|[]],gr(V,E)) :- !. /*
 */ merge_graphs([gr(V1,E1)|G],gr(V3,E3)) :-
        merge_graphs(G,gr(V2,E2)),ord_union(V1,V2,V3),ord_union(E1,E2,E3). /*
```

The predicate 'cartesian_product(Users,Useds,Edges)' creates edges from every element of Users to every element of Useds.

```
 */ cartesian_product([User|Users],Useds,Edges) :-
      product_vertex(User,Useds,Edges1),
      cartesian_product(Users,Useds,Edges2),
      append(Edges1,Edges2,Edges). /*
 */ cartesian_product([],_,[]) :- true. /*
```

The predicate 'product_vertex(User,Useds,Edges)' creates edges from User to every element of Useds.

```
 */ product_vertex(User,[Used|Useds],[User-Used|Edges]) :-
      product_vertex(User,Useds,Edges). /*
 */ product_vertex(_,[],[]) :- true. /*
```

The predicate 'product_graph(+G1,+G2,-G3)' computes G3 as the product of G1 and G2, ie., if there is an edge from U to V in G1 and from V to W in G2, there is an edge from U to W in G3.

```
 */ product_graph([V-S1|G1],G2,[V-S2|G3]) :-
        product_edges(S1,G2,S2),
        product_graph(G1,G2,G3). /*
 */ product_graph([],_,[]). /*
 */ product_edges([V1|S1],G2,S3) :-
        product_edge_set(V1,G2,S2),
        product_edges(S1,G2,S4),
        ord_union(S2,S4,S3). /*
 */ product_edges([],_,[]). /*
 */ product_edge_set(V,[V-E|G2],E) :- !. /*
 */ product_edge_set(V,[_|G2],E) :- product_edge_set(V,G2,E). /*
 */ product_edge_set(_,[],[]). /*
```

The predicate 'graph_union(+G1,+G2,-G3)' computes G3 as the union of graphs G1 and G2.

```
 */ graph_union([],G,G) :- !. /*
 */ graph_union(G,[],G) :- !. /*
 */ graph_union([V-E1|G1],[V-E2|G2],[V-E3|G3]) :- !,
          ord_union(E1,E2,E3),
          graph_union(G1,G2,G3). /*
 */ graph_union([V1-E1|G1],[V2-E2|G2],[V1-E1|G3]) :- V1@<V2,!,
          graph_union(G1,[V2-E2|G2],G3). /*
 */ graph_union([V1-E1|G1],[V2-E2|G2],[V2-E2|G3]) :-
          graph_union([V1-E1|G1],G2,G3). /*
```

The predicate 'reachable_subgraph(+Sources,+GraphIn,-GraphOut)' sets GraphOut to the subgraph of GraphIn that is reachable from the vertices in Sources.

```
 */ reachable_subgraph(Sources,Graph,SubGraph) :-
       sort(Sources,Sorted),
       reachable_vertices(Sorted,Graph,Vertices),
       subgraph(Graph,Vertices,SubGraph). /*
 */ reachable_vertices(Known,Graph,VerticesOut) :-
       successor_vertices([],Known,Graph,Succs),
       (ord_subset(Succs,Known) -> VerticesOut = Known;
        ord_union(Succs,Known,New),
        reachable_vertices(New,Graph,VerticesOut)
       ). /*
 */ successor_vertices(Known,[Vertex|Vertices],[Vertex-Edges|Graph],Succs) :-
       !,ord_union(Edges,Known,New),
       successor_vertices(New,Vertices,Graph,Succs). /*
 */ successor_vertices(Known,[Vertex1|Vertices],[Vertex2-Edges|Graph],Succs) :-
       Vertex1@<Vertex2,!,
       successor_vertices(Known,Vertices,[Vertex2-Edges|Graph],Succs). /*
 */ successor_vertices(Known,Vertices,[_|Graph],Succs) :-
       successor_vertices(Known,Vertices,Graph,Succs). /*
 */ successor_vertices(Known,[],_,Known). /*
```

The original predicate was:

```
    reachable_subgraph(Defs,Graph,Graph1) :-
       reachable_vertex(Defs,Graph,Defs,Reachable),
       sort(Reachable,Defs1),
       ord_union([Defs,Defs1],Defs2),
       subgraph(Graph,Defs2,Graph1).
```

The predicate 'normalise_graph(+GIn,-GOut)' converts the unsorted graph GIn, possibly containing duplicate vertices and edges, into the sorted graph GOut.

```
 */ normalise_graph(GIn,GOut) :- sort(GIn,G),normalise_vertex(G,GOut). /*
 */ normalise_vertex([],[]). /*
 */ normalise_vertex([V-E1,V-E2|G1],G2) :- !,
      sort(E1,E3),sort(E2,E4),ord_union(E3,E4,E),
      normalise_vertex([V-E|G1],G2). /*
 */ normalise_vertex([V-E1|G1],[V-E2|G2]) :-
      sort(E1,E2),
      normalise_vertex(G1,G2). /*
```

The predicate 'simple_edges(+Edges1,+Edges2,-Simples)' succeeds if Simples is the set of all simple edges in Edges1, i.e., all those that do not have a composite path in Edges2.

```
 */ simple_edges([],_,[]) :- !. /*
 */ simple_edges([U-V|Rest],Edges,Simples) :-
       composite(U,Edges,V),!,simple_edges(Rest,Edges,Simples). /*
 */ simple_edges([U-V|Rest],Edges,[U-V|Simples]) :-
       simple_edges(Rest,Edges,Simples). /*
```

The predicate 'composite(+Vertex,+Edges,?Descendant)' succeeds if there is a path of length > 1 from Vertex to Descendant in the acyclic set of edges, Edges.

```
 */ composite(Vertex,Edges,Descendant) :-
         successor(Vertex,Edges,Successor),successor(Successor,Edges,Successor1),
         descendant(Successor1,Edges,Descendant). /*
```

The predicate 'descendant(+Vertex,+Edges,?Descendant)' succeeds if there is a path of length >= 0 from Vertex to Descendant in the acyclic set of edges, Edges.

```
 */ descendant(Vertex,_,Vertex) :- true. /*
 */ descendant(Vertex,Edges,Descendant) :-
         successor(Vertex,Edges,Successor),
         descendant(Successor,Edges,Descendant). /*
```

The predicate 'successor(+Vertex,+Edges,?Descendant)' succeeds if there is a path of length 1 from Vertex to Descendant in the set Edges.

```
 */ successor(Vertex,[Vertex-Successor|_],Successor) :- true. /*
 */ successor(Vertex,[_|Rest],Successor) :- successor(Vertex,Rest,Successor). /*
```

The predicate 'top_sort(+Graph,-Sorted)' finds a topological ordering of a Graph and returns the ordering as a list of Sorted vertices. Fails iff no ordering exists, i.e., iff the graph contains cycles. Approx O(N log N) time.

```
 */ top_sort(Graph,Sorted) :-
        fanin_counts(Graph,Counts),
        get_top_elements(Counts,Top,0,I),
        ord_list_to_assoc(Counts,Map),
        top_sort_1(Top,I,Map,Sorted). /*
 */ top_sort_1([],0,_,[]). /*
 */ top_sort_1([V-VN|Top0],I,Map0,[V|Sorted]) :-
         dec_counts(VN,I,J,Map0,Map,Top0,Top),
         top_sort_1(Top,J,Map,Sorted). /*
 */ dec_counts([],I,I,Map,Map,Top,Top). /*
 */ dec_counts([N|Ns],I,K,Map0,Map,Top0,Top) :-
         get_assoc(N,Map0,NN-C0,Map1,NN-C),C is C0-1,
         (C=:=0 -> (J is I-1,Top1 = [N-NN|Top0]); (J=I,Top1=Top0)),
         dec_counts(Ns,J,K,Map1,Map,Top1,Top). /*
 */ get_top_elements([],[],I,I). /*
 */ get_top_elements([V-(VN-C)|Counts],Top0,I,K) :-
          (C=:=0 -> (J=I,Top0=[V-VN|Top1]); (J is I+1,Top0=Top1)),
          get_top_elements(Counts,Top1,J,K). /*
 */ fanin_counts(Graph,Counts) :- transpose_edges(Graph,Edges0,[]),
          keysort(Edges0,Edges),
          fanin_counts(Graph,Edges,Counts). /*
 */ fanin_counts([],[],[]). /*
 */ fanin_counts([V-VN|Graph],Edges,[V-(VN-C)|Counts]) :-
       fanin_counts(Edges,V,0,C,Edges1),
       fanin_counts(Graph,Edges1,Counts). /*
 */ fanin_counts([V-_|Edges0],V0,C0,C,Edges) :- V==V0,!,
      C1 is C0+1,fanin_counts(Edges0,V0,C1,C,Edges). /*
 */ fanin_counts(Edges,_,C,C,Edges). /*
```

The predicate 'get_assoc(+Key,+Assoc,?Value)' assumes that Assoc is a proper "assoc" tree. It is true when Key is identical to (==) one of the keys in Assoc, and Value unifies with the associated value.

```
 */ get_assoc(Key,t(K,V,L,R),Val) :-
        compare(Rel,Key,K),
        get_assoc_rel(Rel,Key,V,L,R,Val). /*
 */ get_assoc_rel(=,_,Val,_,_,Val). /*
 */ get_assoc_rel(<,Key,_,Tree,_,Val) :- get_assoc(Key,Tree,Val). /*
 */ get_assoc_rel(>,Key,_,_,Tree,Val) :- get_assoc(Key,Tree,Val). /*
```

The predicate 'get_assoc(+Key,+OldAssoc,?OldValue,?NewAssoc,?NewValue)' is true when OldAssoc and NewAssoc are "assoc" trees of the same shape having the same elements except that the value for Key in OldAssoc is OldValue and the value for Key in NewAssoc is NewValue.

```
*/ get_assoc(Key,t(K0,V0,L0,R0),Val0,t(K,V,L,R),Val) :-
        compare(Rel,Key,K0),
        get_assoc_rel(Rel,Key,K0,V0,L0,R0,Val0,K,V,L,R,Val). /*
*/ get_assoc_rel(=,_,K,Val0,L,R,Val0,K,Val,L,R,Val). /*
*/ get_assoc_rel(<,Key,K,V,Tree0,R,Val0,K,V,Tree,R,Val) :-
      get_assoc(Key,Tree0,Val0,Tree,Val). /*
*/ get_assoc_rel(>,Key,K,V,L,Tree0,Val0,K,V,L,Tree,Val) :-
      get_assoc(Key,Tree0,Val0,Tree,Val). /*
```

The predicate 'ord_list_to_assoc(+List,?Assoc)' is true when List is a proper list of Key-Val pairs (keysorted) and Assoc is an association tree specifying the same finite function from Keys to Values.

```
*/ ord_list_to_assoc(List,Assoc) :-
     length(List,N),
     list_to_assoc(N,List,Assoc,[]). /*
*/ list_to_assoc(0,List,t,List) :- !. /*
*/ list_to_assoc(N,List,t(Key,Val,L,R),Rest) :-
       A is (N-1) >> 1,
       Z is (N-1)-A,
       list_to_assoc(A,List,L,[Key-Val|More]),
       list_to_assoc(Z,More,R,Rest). /*
```

The predicate 'ord_del_element(+Set1,+Element,?Set2)' is true when Set2 is Set1 but with Element removed.

```
*/ ord_del_element([],_,[]). /*
*/ ord_del_element([Head|Tail],Element,Set) :-
        compare(Order,Head,Element),
        ord_del_element_rel(Order,Head,Tail,Element,Set). /*
*/ ord_del_element_rel(<,Head,Tail,Element,[Head|Set]) :-
            ord_del_element(Tail,Element,Set). /*
*/ ord_del_element_rel(=,_,Tail,_,Tail). /*
*/ ord_del_element_rel(>,Head,Tail,_,[Head|Tail]). /*
```

The predicate 'ord_intersection(+Set1,+Set2,?Intersection)' is true when Intersection is the ordered representation of Set1 and Set2, provided that Set1 and Set2 are ordered sets.

```
*/ ord_intersection([],_,[]) :- !. /*
*/ ord_intersection(_,[],[]) :- !. /*
*/ ord_intersection([Head1|Tail1],[Head2|Tail2],Intersection) :-
          compare(Order,Head1,Head2),
          ord_intersection_rel(Order,Head1,Tail1,Head2,Tail2,Intersection). /*
*/ ord_intersection_rel(<,_,[],_,_,[]) :- !. /*
*/ ord_intersection_rel(<,_,[Head1|Tail1],Head2,Tail2,Intersection) :-
          compare(Order,Head1,Head2),
          ord_intersection_rel(Order,Head1,Tail1,Head2,Tail2,Intersection). /*
*/ ord_intersection_rel(=,Head,Tail1,_,Tail2,[Head|Intersection]) :-
          ord_intersection(Tail1,Tail2,Intersection). /*
*/ ord_intersection_rel(>,_,_,_,[],[]) :- !. /*
*/ ord_intersection_rel(>,Head1,Tail1,_,[Head2|Tail2],Intersection) :-
          compare(Order,Head1,Head2),
          ord_intersection_rel(Order,Head1,Tail1,Head2,Tail2,Intersection). /*
```

The predicate ord_subset(+Set1,+Set2) is true when every element of the ordered set Set1 appears in the ordered set Set2.

```
*/ ord_subset([],_). /*
*/ ord_subset([Head1|Tail1],[Head2|Tail2]) :-
            compare(Order,Head1,Head2),
            ord_subset_rel(Order,Head1,Tail1,Tail2). /*
*/ ord_subset_rel(=,_,Tail1,Tail2) :- ord_subset(Tail1,Tail2). /*
*/ ord_subset_rel(>,Head1,Tail1,[Head2|Tail2]) :-
           compare(Order,Head1,Head2),
           ord_subset_rel(Order,Head1,Tail1,Tail2). /*
```

The predicate 'ord_union(+Set1,+Set2,?Union)' is true when Union is the union of Set1 and Set2. Note that when something occurs in both sets, we want to retain only one copy.

```
*/ ord_union([],Set2,Set2) :- !. /*
*/ ord_union(Set1,[],Set1) :- !. /*
*/ ord_union([Head1|Tail1],[Head2|Tail2],Union) :-
            compare(Order,Head1,Head2),
            ord_union_rel(Order,Head1,Tail1,Head2,Tail2,Union). /*
*/ ord_union_rel(<,Head0,[],Head2,Tail2,[Head0,Head2|Tail2]) :- !. /*
*/ ord_union_rel(<,Head0,[Head1|Tail1],Head2,Tail2,[Head0|Union]) :-
            compare(Order,Head1,Head2),
            ord_union_rel(Order,Head1,Tail1,Head2,Tail2,Union). /*
*/ ord_union_rel(=,Head,Tail1,_,Tail2,[Head|Union]) :-
            ord_union(Tail1,Tail2,Union). /*
*/ ord_union_rel(>,Head1,Tail1,Head0,[],[Head0,Head1|Tail1]) :- !. /*
*/ ord_union_rel(>,Head1,Tail1,Head0,[Head2|Tail2],[Head0|Union]) :-
            compare(Order,Head1,Head2),
            ord_union_rel(Order,Head1,Tail1,Head2,Tail2,Union). /*
```

The predicate 'ord_union(+Sets,?Union)' is true when Union is the union of all the sets in Sets.

```
 */ ord_union([],Union) :- !,Union = []. /*
 */ ord_union(Sets,Union) :-
            length(Sets,NumberOfSets),
            ord_union_all(NumberOfSets,Sets,Union,[]). /*
 */ ord_union_all(1,[Set|Sets],Set,Sets) :- !. /*
 */ ord_union_all(2,[Set,Set2|Sets],Union,Sets) :- !, ord_union(Set,Set2,Union). /*
 */ ord_union_all(N,Sets0,Union,Sets) :-
            A is N>>1,ord_union_all(A,Sets0,X,Sets1),
            Z is N-A,ord_union_all(Z,Sets1,Y,Sets),
            ord_union(X,Y,Union). /*
```

The predicate 'ord_union(+Set1,+Set2,?Union,?New)' is true when Union is the union of Set1 and Set2, and New is the difference between Set2 and Set1. This is useful if you are accumulating members of a set and you want to process new elements as they are added to the set.

```
 */ ord_union([],Set,Set,Set) :- !. /*
 */ ord_union(Set,[],Set,[]) :- !. /*
 */ ord_union([O|Os],[N|Ns],Set,New) :-
            compare(C,O,N),
            ord_union_rel(C,O,Os,N,Ns,Set,New). /*
 */ ord_union_rel(<,O,[],N,Ns,[O,N|Ns],[N|Ns]) :- !. /*
 */ ord_union_rel(<,O1,[O|Os],N,Ns,[O1|Set],New) :-
           compare(C,O,N),
           ord_union_rel(C,O,Os,N,Ns,Set,New). /*
 */ ord_union_rel(=,_,Os,N,Ns,[N|Set],New) :- ord_union(Os,Ns,Set,New). /*
 */ ord_union_rel(>,O,Os,N,[],[N,O|Os],[N]) :- !. /*
 */ ord_union_rel(>,O,Os,N1,[N|Ns],[N1|Set],[N1|New]) :-
           compare(C,O,N),
           ord_union_rel(C,O,Os,N,Ns,Set,New). /*
 */ portray(X) :- nl,pretty_term(0,X),write('.'),nl. /*
```

The predicate 'pretty(X)' is logically equivalent to 'writeq(X),write('.')', i.e., it writes a term in a form suitable for reading by 'read(X)'. However, its output is better formatted and more user-friendly.

```
 */ pretty(X) :- nl,pretty_term(0,X),write('.'),nl,nl. /*
 */ pretty_term(N,X) :- var(X),!,write(X). /*
 */ pretty_term(N,lex(X,Y)) :- !,write(lex(X,Y)). /*
 */ pretty_term(N,state(X,Y)) :- !,write(state(X,Y)). /*
 */ pretty_term(N,param(X,Y)) :- !,write(param(X,Y)). /*
 */ pretty_term(N,local(X,Y)) :- !,write(local(X,Y)). /*
 */ pretty_term(N,event(X,Y)) :- !,write(event(X,Y)). /*
 */ pretty_term(N,dd(X,Y,Z)) :- !,write(dd(X,Y,Z)). /*
 */ pretty_term(N,gr(V,E)) :- !,write('gr('),pretty_term(N,V),write(','),nl,
          N1 is N+3,tab(N1),pretty_term(N1,E),write(')'). /*
 */ pretty_term(N,[]) :- !,write('[]'). /*
 */ pretty_term(N,[H|T]) :-
          !,write('['),N1 is N+1,pretty_term(N1,H),pretty_tail(N1,T),write(']'). /*
 */ pretty_term(N,X) :- atom(X),!,writeq(X). /*
 */ pretty_term(N,X) :- integer(X),!,write(X). /*
 */ pretty_term(N,X/Y) :- simple_term(Y),!,
          pretty_term(N,X),write('/'),pretty_term(N,Y). /*
 */ pretty_term(N,X/Y) :- !,pretty_term(N,X),write('/'),nl,
          N1 is N+1,tab(N1),pretty_term(N1,Y). /*
 */ pretty_term(N,X-Y) :- simple_term(Y),!,
         pretty_term(N,X),write('-'),pretty_term(N,Y). /*
 */ pretty_term(N,X-Y) :- !,pretty_term(N,X),write('-'),nl,
          N1 is N+1,tab(N1),pretty_term(N1,Y). /*
 */ pretty_term(N,(X,Y)) :-
         !,write('('),N1 is N+1,pretty_term(N1,X),write(','),
         pretty_term(N1,Y),write(')'). /*
 */ pretty_term(N,E) :- compound(E),!,E=..[F|Args],
         writeq(F),write('('),N1 is N+1,pretty_list(N1,Args),write(')'). /*
 */ pretty_tail(N,X) :- var(X),!,write(','),write(X). /*
 */ pretty_tail(N,[]) :- !. /*
 */ pretty_tail(N,T)  :- simple_list(T),!,write(','),pretty_list(N,T). /*
 */ pretty_tail(N,T)  :- write(','),nl,tab(N),pretty_list(N,T). /*
 */ pretty_list(N,X)      :- var(X),!,write(X). /*
 */ pretty_list(N,[])     :- !. /*
 */ pretty_list(N,[H|T]) :- !,pretty_term(N,H),pretty_tail(N,T). /*
 */ simple_list(X) :- var(X),!. /*
 */ simple_list([]) :- !. /*
 */ simple_list([T|[]]) :- simple_term(T),!. /*
 */ simple_list([H,T|[]]) :- simple_term(H),simple_term(T). /*
 */ simple_term(X) :- var(X),!. /*
 */ simple_term([]) :- !. /*
 */ simple_term(X) :- atomic(X),!. /*
 */ simple_term(X-Y) :- atomic(X),atomic(Y),!. /*
 */ simple_term(X/Y) :- simple_term(X),simple_term(Y),!. /*
 */ simple_term((X,Y)) :- simple_term(X),simple_term(Y),!. /*
```

The predicate incidence_matrix(+Graph) succeeds by displaying a form of incidence matrix of Graph. Each vertex is allocated a column. The diagonal is marked '*', or '@' if there is a loop. Empty cells are marked '.|_'. Edges are marked '+'. Each row represents a vertex and its edges.

```
 */ incidence_matrix(Graph) :-
          vertices(Graph,Vertices),
          length(Vertices,N),
          N2 is N+1,nl,
          incidence(1,N2,Graph,Vertices). /*
 */ incidence(M,N,[Vertex-Edges|Graph],Vertices) :-
        display_row(1,M,N,Vertex,Vertices,Edges),
        (M==5 -> M1=1; M1 is M+1),
        incidence(M1,N,Graph,Vertices). /*
 */ incidence(_,_,[],_) :- nl. /*
 */ display_row(K,M,N,Vertex,[Vertex|Vertices],[Vertex|Edges]) :- !,
         (K==5 -> K1=1; K1 is K+1),
        write('@'),display_rest(K1,M,N,Vertex,Vertices,Edges). /*
 */ display_row(K,M,N,Vertex,[Vertex|Vertices],Edges) :- !,
         (K==5 -> K1=1; K1 is K+1),
        write('o'),display_rest(K1,M,N,Vertex,Vertices,Edges). /*
 */ display_row(K,M,N,Vertex,[Edge|Vertices],[Edge|Edges]) :- !,
         (K==5 -> K1=1; K1 is K+1),
        write('^'),display_row(K1,M,N,Vertex,Vertices,Edges). /*
 */ display_row(K,M,N,Vertex,[_|Vertices],Edges) :- !,
         (K==5 -> K1=1; K1 is K+1),
         (K==5 -> write('|'); M==5 -> write('_'); write('.')),
        display_row(K1,M,N,Vertex,Vertices,Edges). /*
 */ display_rest(K,M,N,Vertex,[Edge|Vertices],[Edge|Edges]) :- !,
         (K==5 -> K1=1; K1 is K+1),
        write('v'),display_rest(K1,M,N,Vertex,Vertices,Edges). /*
 */ display_rest(K,M,N,Vertex,[_|Vertices],Edges) :- !,
         (K==5 -> K1=1; K1 is K+1),
         (K==5 -> write('|'); M==5 -> write('_'); write('.')),
        display_rest(K1,M,N,Vertex,Vertices,Edges). /*
 */ display_rest(K,M,N,Vertex,[],[]) :- !,write(' '),pretty_term(N,Vertex),nl. /*
 */ display_rest(K,M,N,Vertex,[],Edges) :-
        write(' '),pretty_term(N,Vertex),write(' bug:'),write(Edges),nl. /*
 */ edge_matrix(Edges) :-
       vertices_edges_to_graph([],Edges,Graph1),
       map_to_lexes(Graph1,Graph),
       incidence_matrix(Graph). /*
```

The predicate 'map_to_lexes(+GraphIn,-GraphOut)' converts a dynamic use-definition graph into its equivalent lexical use-definition graph.

```
 */ map_to_lexes(GraphIn,GraphOut) :-
          vertices(GraphIn,Procs),
          edges(GraphIn,Edges),
          map_processes(Procs,Procs1),
          map_edges(Edges,Edges1),
          vertices_edges_to_graph(Procs1,Edges1,GraphOut). /*
 */ map_processes([],[]) :- true. /*
 */ map_processes([ProcIn|ProcsIn],[ProcOut|ProcsOut]) :-
          map_process(ProcIn,ProcOut),
          map_processes(ProcsIn,ProcsOut). /*
 */ map_edges([],[]) :- true. /*
 */ map_edges([UsedIn-UserIn|EdgesIn],[UsedOut-UserOut|EdgesOut]) :-
        map_process(UsedIn,UsedOut),
        map_process(UserIn,UserOut),
        map_edges(EdgesIn,EdgesOut). /*
 */ map_process((Degree,DefsIn),(Degree,DefsOut)) :- !,
         map_defs(DefsIn,Defs),
         sort(Defs,DefsOut). /*
 */ map_process([DefsIn],DefsOut) :- !,
         map_defs([DefsIn],Defs),
         sort(Defs,DefsOut). /*
 */ map_process(DefIn,lex(Vars,Lex)) :- !,
         map_to_ids_lexes(DefIn,Vars,[Lex|Lexes]). /*
 */ map_defs([],[]) :- true. /* done
 */ map_defs([DefIn|DefsIn],[lex(Vars,Lex)|DefsOut]) :- !,
       map_to_ids_lexes(DefIn,Vars,[Lex|Lexes]),
       map_defs(DefsIn,DefsOut). /* other
 */ map_to_ids_lexes(dd([],_,_),[],[]) :- true. /*
 */ map_to_ids_lexes(dd([(Id,Lex,_)|Subs],Flag,Freq),[Id|Ids],[Lex|Lexes]) :-
       map_to_ids_lexes(dd(Subs,Flag,Freq),Ids,Lexes). /*
 */ map_to_names((Degree,Defs),(Degree,Names)) :-
        map_to_ids(Defs,Ids),
        sort(Ids,Names). /*
 */ map_to_ids([],[]) :- true. /*
 */ map_to_ids([dd(Var,_,_)|Defs],[Id|Ids]) :-
        map_var_to_ids(Var,Id),
        map_to_ids(Defs,Ids). /*
 */ map_var_to_ids([],[]) :- true. /*
 */ map_var_to_ids([(*,_,_)|Subs],[*|Ids]) :- map_var_to_ids(Subs,Ids). /*
 */ map_var_to_ids([(_/_/Id,_,_)|Subs],[Id|Ids]) :- map_var_to_ids(Subs,Ids). /*
The predicate 'dummy_list(+X,+Y,-Z)' succeeds when Z is a list of X's of the same length as Y.
 */ dummy_list(_,[],[]). /*
 */ dummy_list(X,[_|T],[X|U]) :- dummy_list(X,T,U). /*
The predicate 'unique(-Num)' succeeds by assigning a new serial number to 'Num', increasing 'Num' by 1 on each
call.
 */ unique(N) :- retract(current_num(N1)),!,N is N1 + 1,asserta(current_num(N)). /*
 */ unique(1) :- asserta(current_num(1)). /*
The predicate 'unique(+Prefix,-Num)' succeeds by assigning a new serial number to 'Num', increasing 'Num' by 1 on
each call.  There is a separate sequence for each value of Prefix.
 */ unique(Prefix,N) :- retract(current_num(Prefix,N1)),!,
        N is N1 + 1,asserta(current_num(Prefix,N)). /*
 */ unique(Prefix,1) :- asserta(current_num(Prefix,1)). /*
The predicate 'member(+Element,+List)' succeeds if Element is in the list List.
 */ member(H,[H|_]) :- !. /*
 */ member(H,[_|T]) :- member(H,T). /*
The predicate 'append(?A,?B,?C)' succeeds if list C contains the terms of list A followed by the terms of list B.
 */ append(L,[],L). /*
 */ append([],L,L). /*
 */ append([W|L1],L2,[W|L3]) :- append(L1,L2,L3). /*
End of shared predicates. */
```

13.3 The Dependence Analyser

The *Analyser* may be invoked by a goal of the form:

```
analyse(Tree,Graph).
```

where *Tree* is a file containing the abstract syntax tree generated by the *Parser*, and *Graph* is the file that should contain the hard and soft SDG's generated by the *Analyser*.

```
/* The Analyser.
The purpose of this program is to read the syntax tree of a specification and convert it into a use-definition
graph.
 */ analyse(Input,Output) :-
       read_spec(Input,Spec),
       analyse_system(Spec,Decls,FinalDefs,Hard,Soft),
       write_graph(Output,Decls,FinalDefs,Hard,Soft),!. /*
 */ read_spec(Input,Spec) :- seeing(User),see(Input),read(Spec),seen,see(User). /*
 */ write_graph(Output,Decls,FinalDefs,Hard,Soft) :-
          telling(User),tell(Output),
          pretty(Decls),
          pretty(FinalDefs),
          pretty(Hard),
          pretty(Soft),
          incidence_matrix(Hard),
          incidence_matrix(Soft),
          told,tell(User). /*
The only important parts of the system specification are the variable declarations and the event specifications,
since these are the only things that create definitions.  A separate list is made of the types of all variables,
so that the degree of parallelism can be tested later.
```

```
 */ analyse_system(system(_,_,_,StateVars,Events),Decls,FinalDefs,UGraph,LGraph) :-
        'system$push$display'(message,left,'Analysing dependences...','','',''),
        abolish(current_num,2),
        analyse_state_defs(StateVars,[],StateDefs,Decls1),
        sort(StateDefs,InitialDefs),
        analyse_events(Events,InitialDefs,FinalDefs,Hard,Soft,Decls2),
        vertices_edges_to_graph([],Hard,UGraph),
        vertices_edges_to_graph([],Soft,LGraph),
        append(Decls1,Decls2,Decls),
        'system$pop$display'(message). /*
```

The state variable definitions are formed into a single list. The codomain of a state variable is relevant to type-checking, but not to use-definition analysis. The initial definition of each state variable is constructed from the variable name, a dummy subscript list, and a tag.

```
 */ analyse_state_defs([state(Id,Codom)|Vars],DefsIn,DefsOut,[(Id,Codom)|Decls]) :-
        analyse_declaration((Id,Codom),DefsIn,Defs1),
        analyse_state_defs(Vars,Defs1,DefsOut,Decls). /*
 */ analyse_state_defs([],Defs,Defs,[]). /*
```

The predicate 'analyse_declaration(([+Id|+Dom],Codom),+DefsIn,-DefsOut,-Decl)' succeeds when a new definition has been created for variable 'Id' with domain 'Dom' (empty for a simple variable). The definition has the form 'dd([(Id,N,Tag),(*,0,0),..],Flag,Freq)'. If the same identifier is already declared, with the same number of subscripts, its old definition is lost.

```
 */ analyse_declaration((lex([Id|Dom],N),_),DefsIn,[NewDef|DefsIn]) :-
      unique(Id,Tag),
      dummy_list(('*',0,0),Dom,Stars),
      NewDef=dd([(Id,N,Tag)|Stars],rd,[]). /*
```

Since there may be several events, all their definitions are appended to form a single list. Since events cannot see one another's definitions, they all begin with the same set of global definitions.

```
 */ analyse_events([Event|Events],DefsIn,DefsOut,Hard,Soft,Decls) :-
        !,analyse_event(Event,DefsIn,DefsOut1,Hard1,Soft1,Decls1),
        analyse_events(Events,DefsIn,DefsOut2,Hard2,Soft2,Decls2),
        append(Decls1,Decls2,Decls),
        ord_union([DefsOut1,DefsOut2],DefsOut),
        ord_union(Soft1,Soft2,Soft),
        ord_union(Hard1,Hard2,Hard). /*
 */ analyse_events([],DefsIn,DefsIn,[],[],[]) :- true. /*
```

Each event makes initial definitions of its parameters and local variables, and creates definitions through assignment statements, etc.

```
 */ analyse_event(event(Event,Params,Locals,Stmts),InitialDefs,
                FinalDefs,Hard,Soft,Decls) :-
      analyse_param_defs(Params,InitialDefs,Defs1,Decls1),
      analyse_local_defs(Locals,Defs1,Defs2,Decls2),
      sort(Defs2,DeclDefs),
      analyse_stmt(([],[]),Stmts,DeclDefs,Defs4,Hard1,Soft1,Decls3),
      drop_all_locals([],Defs4,Defs5,Hard2,Soft2),
      link_globals_to_decls(Defs5,FinalDefs,Hard3,Soft3),
      ord_union([Decls1,Decls2,Decls3],Decls),
      ord_union([Hard1,Hard2,Hard3],Hard),
      ord_union([Soft1,Soft2,Soft3],Soft). /*
```

The parameter definitions are added to the global definitions.

```
 */ analyse_param_defs([param(Id,Codom)|Vars],DefsIn,DefsOut,[(Id,Codom)|Decls]) :-
        analyse_declaration((Id,Codom),DefsIn,Defs1),
        analyse_param_defs(Vars,Defs1,DefsOut,Decls). /*
 */ analyse_param_defs([],Defs,Defs,[]). /*
```

The local definitions are added to the parameter definitions and global definitions.

```
 */ analyse_local_defs([local(Id,Codom)|Vars],DefsIn,DefsOut,[(Id,Codom)|Decls]) :-
        analyse_declaration((Id,Codom),DefsIn,Defs1),
        analyse_local_defs(Vars,Defs1,DefsOut,Decls). /*
 */ analyse_local_defs([],Defs,Defs,[]). /*
```

The predicate 'analyse_stmt(+Context,+Statement,+DefsIn,-DefsOut,-Graph,-Decls)' accepts a 'Context' (consisting of the definitions on which the statement list is conditional), a 'Statement', and 'DefsIn', a list of definitions valid on entry to the statement list. It produces 'DefsOut', an updated list of valid definitions, and 'Graph', the new vertices and edges of the use-definition graph created by the statement list.

In a sequence of statements, the 'DefsOut' definitions of the first statement become the 'DefsIn' definitions of the second, and so on. The 'Graph' of a sequence is the union of the individual graphs.

```
 */ analyse_stmt(_,[],DefsIn,DefsIn,[],[],[]) :- !. /* empty statement list
 */ analyse_stmt((Freq,Context),[Stmt|Stmts],DefsIn,DefsOut,Hard,Soft,Decls) :- !,
      analyse_stmt((Freq,Context),Stmt,DefsIn,Defs1,Hard1,Soft1,Decls1),
      analyse_stmt((Freq,Context),Stmts,Defs1,DefsOut,Hard2,Soft2,Decls2),
      append(Decls1,Decls2,Decls),
      ord_union([Hard1,Hard2],Hard),
      ord_union([Soft1,Soft2],Soft). /* non-empty list
```

An assignment statement creates a new definition of its LHS variable, and removes any existing definitions of the same variable. It begins by analysing the RHS expression, after which all RHS variables should have definitions. Then it kills all definitions of its LHS variable. It then creates a subgraph for its LHS variable. Finally, it links the definitions in the RHS expression and the Context to the definition of the LHS variable.

```
 */ analyse_stmt((Freq,Context),assign(Var,Expn),DefsIn,DefsOut,Hard,Soft,[]) :- !,
          analyse_expn(Freq,Expn,DefsIn,Defs1,ExprDefs,Hard1,Soft1),
          Var=lex([Id|Subs],_),
          drop_var_def([Id|Subs],Defs1,Defs2),
          analyse_var_defn(Freq,Var,Defs2,VarDef,Hard3,Soft3),
          cartesian_product([VarDef],ExpnDefs,Hard4),
          cartesian_product([VarDef],Context,Hard5),
          ord_union([[VarDef],Defs2],DefsOut),
          ord_union([Hard1,Hard3,Hard4,Hard5],Hard),
          ord_union([Soft1,Soft3],Soft). /* assignment
```

The 'call' statement is like an assignment, but any existing definitions of the same call need to stay live, not get dropped.

```
 */ analyse_stmt((Freq,Context),call(Proc,Expn),DefsIn,DefsOut,Hard,Soft,Decls) :-
        Proc=lex([Id|[]],N),find_live_def([Id|[]],DefsIn,_),!,
        Expn1=[Proc|Expn],
        analyse_stmt((Freq,Context),assign(Proc,Expn1),DefsIn,
                     DefsOut,Hard,Soft,Decls). /*
 */ analyse_stmt((Freq,Context),call(Proc,Expn),DefsIn,DefsOut,Hard,Soft,Decls) :- !,
        analyse_stmt((Freq,Context),assign(Proc,Expn),DefsIn,
                     DefsOut,Hard,Soft,Decls). /*
```

An 'if' statement is handled by finding the definitions used in its conditional expression, and adding these to the conditional context. Its 'true' and 'false' branches are analysed recursively, and their results are merged. Why isn't Context passed into 'analyse_expn'? If it were, array references used in the conditional expression would depend on enclosing conditions, whereas there is no reason that they can't be inspected unconditionally.

```
 */ analyse_stmt((Freq,Context),if(Var,Expn,True,False),DefsIn,
                  DefsOut,Hard,Soft,Decls) :-
          !,analyse_stmt((Freq,Context),assign(Var,Expn),DefsIn,
                         Defs1,Hard1,Soft1,_),
          Var=lex(Id,_),find_live_def(Id,Defs1,ExpDef),
          analyse_stmt((Freq,[ExpDef]),True,Defs1,Defs2,Hard2,Soft2,Decls1),
          analyse_stmt((Freq,[ExpDef]),False,Defs1,Defs3,Hard3,Soft3,Decls2),
          append(Decls1,Decls2,Decls),
          join_branches(Freq,Defs2,Defs3,DefsOut,Hard4,Soft4),
          ord_union([Hard1,Hard2,Hard3,Hard4],Hard),
          ord_union([Soft1,Soft2,Soft3,Soft4],Soft). /* if statement
```

A 'while' statement is an infinite nest of 'if' statements, but two will do.

```
 */ analyse_stmt((Freq,Context),while(Var,Expn,Body),DefsIn,
                  DefsOut,Hard,Soft,Decls) :- !,
      analyse_stmt((Freq,Context),loop(Var,Expn,Body),
                   DefsIn,DefsOut,Hard,Soft,Decls). /*
```

An 'all' statement is equivalent to a loop containing an assignment.

```
 */ analyse_stmt((Freq,Context),all(Var,Codom,Body),
                  DefsIn,DefsOut,Hard,Soft,[(Var,Codom)|Decls]) :- !,
        analyse_stmt((Freq,Context),loop(Var,[],[assign(Var,[]),Body]),
                     DefsIn,DefsOut,Hard,Soft,Decls). /*
```

A 'for' statement is equivalent to a loop preceded by an assignment.

```
 */ analyse_stmt((Freq,Context),for(Var,Codom,Body),
                  DefsIn,DefsOut,Hard,Soft,[(Var,Codom)|Decls]) :- !,
        analyse_stmt((Freq,Context),[assign(Var,[]),loop(Var,[],Body)],
                     DefsIn,DefsOut,Hard,Soft,Decls). /*
```

A 'loop' statement is a general-purpose statement used by 'while', 'all', and 'for'.

```
 */ analyse_stmt((Freq,Context),loop(Var,Expn,Body),DefsIn,
                  DefsOut,Hard,Soft,Decls) :- !,
      analyse_stmt((Freq1,Context),assign(Var,Expn),DefsIn,Defs1,Hard1,Soft1,_),
      Var=lex(Name,_),find_live_def(Name,Defs1,ExpDef1),
      ExpDef1=dd(Vars,_,_),
      append(Freq,Vars,Freq1),
      analyse_stmt((Freq1,[ExpDef1]),Body,Defs1,Defs2,Hard2,Soft2,Decls),
      analyse_stmt((Freq1,[ExpDef1]),assign(Var,Expn),Defs2,Defs4,Hard4,Soft4,_),
      find_live_def(Name,Defs4,ExpDef2),
      analyse_stmt((Freq1,[ExpDef2]),Body,Defs4,Defs5,Hard5,Soft5,_),
      link_identical_elements(Defs2,Defs5,Hard6,Soft6),
      join_branches(Freq1,Defs1,Defs5,Defs6,Hard7,Soft7),
      drop_loop_var(Freq1,Name,Defs6,DefsOut,Hard8,Soft8),
      ord_union([Hard1,Hard2,Hard4,Hard5,Hard6,Hard7,Hard8],Hard),
      ord_union([Soft1,Soft2,Soft4,Soft5,Soft6,Soft7,Soft8],Soft). /*
```

The 'null' statement has no effect.

```
 */ analyse_stmt(_,null,DefsIn,DefsIn,[],[],[]) :- !. /*
```

A 'return' statement is of no interest except in an event procedure. It has no associated expression. All local variables must be dropped.

```
 */ analyse_stmt((Freq,_),return(Expn),DefsIn,DefsOut,Hard,Soft,[]) :- !,
       drop_all_locals(Freq,DefsIn,DefsOut,Hard,Soft). /*
```

The declare statement introduces local variables, especially within an 'all' loop.

```
 */ analyse_stmt((Freq,Context),declare(Vars,Body),DefsIn,DefsOut,Hard,Soft,Decls) :-
        analyse_local_defs(Vars,DefsIn,Defs1,Decls1),
        analyse_stmt((Freq,Context),Body,Defs1,DefsOut,Hard,Soft,Decls2),
        append(Decls1,Decls2,Decls). /*
```

Anything else is an error:

```
 */ analyse_stmt(Context,Stmt,DefsIn,DefsOut,Hard,Soft,Decls) :-
        write('Unrecognised statement:'),nl,
        pretty(analyse_stmt(Context,Stmt,DefsIn,DefsOut,Hard,Soft,Decls)),nl. /*
```

The predicate 'analyse_var_defn(+Var,+DefsIn,-Def,-Hard,-Soft)' constructs a new definition Def for variable Var, by reference to the set of existing definitions DefsIn.

```
 */ analyse_var_defn(Freq,lex([Id|[]],N),_,VarDef,[],[]) :- !,
      unique(Id,Tag),VarDef=dd([(Id,N,Tag)],wr,Freq). /*
 */ analyse_var_defn(Freq,lex([Id|Subs],N),DefsIn,VarDef,Hard,[]) :-
      unique(Id,Tag),
      lookup_tags(Subs,DefsIn,TaggedSubs),
      VarDef=dd([(Id,N,Tag)|TaggedSubs],wr,Freq),
      live_subscript_defs(Subs,DefsIn,SubDefs),
      cartesian_product([VarDef],SubDefs,Hard). /*
```

The predicate 'analyse_expn(+Freq,+Expn,+DefsIn,-DefsOut,-ExpnDefs,-Hard)' creates the subgraph 'Graph' for the list of variables in 'Expn', with the corresponding list of definitions ExpnDefs. Although an expression does not make any assignments, it may create definitions. If a term such as 'A(i)' appears, it is treated as defining '[A,i]' using '[A,*]' and '[i]', unless there is already a definition of '[A,i]' in 'DefsIn'. Apart from tese additions, DefsOut is a copy of DefsIn.

```
 */ analyse_expn(Freq,[Term|Expn],DefsIn,DefsOut,ExpnDefs,Hard,Soft) :-
       analyse_term(Freq,Term,DefsIn,Defs1,ExpnDefs1,Hard1,Soft1),
       analyse_expn(Freq,Expn,Defs1,DefsOut,ExpnDefs2,Hard2,Soft2),
       ord_union([ExpnDefs1,ExpnDefs2],ExpnDefs),
       ord_union([Hard1,Hard2],Hard),
       ord_union([Soft1,Soft2],Soft). /*
 */ analyse_expn(_,[],DefsIn,DefsIn,[],[],[]) :- true. /*
```

The 'analyse_term' predicate deals with one element of an expression. Simple variables and array elements are dealt with separately.

```
 */ analyse_term(Freq,lex([Id|[]],N),DefsIn,DefsIn,ExpnDefs,Hard,Soft) :- !,
      analyse_simple_use(Freq,lex([Id|[]],N),DefsIn,ExpnDefs,Hard,Soft). /*
 */ analyse_term(Freq,Def,DefsIn,DefsOut,ExpnDefs,Hard,Soft) :-
      analyse_element_use(Freq,Def,DefsIn,DefsOut,ExpnDefs,Hard,Soft). /*
```

The predicate 'analyse_simple_use(+Def,+DefsIn,-ExpnDefs,-Hard,-Soft)' returns the existing definition as the only member of ExpnDefs. If the variable has no live definition there is an error.

```
 */ analyse_simple_use(Freq,lex([Id|[]],N),DefsIn,[OldDef],[],[]) :-
        find_live_def([Id|[]],DefsIn,OldDef),!. /*
 */ analyse_simple_use(Freq,lex([Id|[]],N),DefsIn,[NewDef],[],[]) :-
        Id=output/_/_,!,
        unique(Id,Tag),NewDef=dd([(Id,N,Tag)],rd,Freq). /* used in calls
 */ analyse_simple_use(Freq,lex([Id|[]],N),DefsIn,[NewDef],[],[]) :-
        unique(Id,Tag),NewDef=dd([(Id,N,Tag)],rd,Freq),
        writeq(Id),write(' is not defined in '),nl,pretty(DefsIn). /*
```

The predicate 'analyse_element_use(+Var,+DefsIn,-DefsOut,-ExpnDef,-Graph)' constructs a subgraph corresponding to a RHS array element reference. If the element has an existing definition, the situation is similar to that for a simple variable. If there is no existing definition, the indexed variable uses the existing definition of the array (GenericDef), which in turn uses the definitions of the subscripts (SubDefs). DefsOut is formed from DefsIn by adding the new element definition.

```
 */ analyse_element_use(Freq,lex([Id|Subs],N),DefsIn,DefsIn,[OldDef],[],[]) :-
          find_live_def([Id|Subs],DefsIn,OldDef),!. /*
 */ analyse_element_use(Freq,lex([Id|Subs],N),DefsIn,DefsOut,[Element],Hard,Soft) :-
          drop_alias(Freq,Id,DefsIn,Defs1,Hard1,Soft1),
          generic_def([Id|Subs],Defs1,GenericDef),
          unique(Id,Tag),
          lookup_tags(Subs,Defs1,TaggedSubs),
          Element=dd([(Id,N,Tag)|TaggedSubs],rd,Freq),
          live_subscript_defs(Subs,Defs1,SubDefs),
          cartesian_product([Element],SubDefs,Hard2),
          cartesian_product([Element],[GenericDef],Soft2),
          ord_union([[Element],Defs1],DefsOut),
          ord_union([Hard1,Hard2],Hard),
          ord_union([Soft1,Soft2],Soft). /*
```

The predicate 'drop_alias(+Freq,+Generic,+DefsIn,-DefsOut,-Hard,-Soft)' sets DefsOut to DefsIn, less the definition of any element related to Generic present in DefsIn.

```
 */ drop_alias(Freq,Id,DefsIn,DefsOut,Hard,Soft) :-
        find_alias(Id,DefsIn,Element),!,
        array_uses_element(Freq,Element,DefsIn,DefsOut,Hard,Soft). /*
 */ drop_alias(_,_,DefsIn,DefsIn,[],[]) :- true. /*
```

The predicate 'find_alias(+Generic,+DefsIn,-Element)' succeeds if it sets Element to the definition of the first (and only) element of Generic present in DefsIn.

```
 */ find_alias(Id,[Generic|DefsIn],Element) :-
         Generic=dd([(Id,_,_),(*,0,0)|_],_,_),!,
         find_alias(Id,DefsIn,Element). /* ignore the generic itself
 */ find_alias(Id,[Element|_],Element) :-
         Element=dd([(Id,_,_)|_],_,_),!. /* Ids match, but not subscripts, QED.
 */ find_alias(Id,[_|DefsIn],Element) :-
         find_alias(Id,DefsIn,Element). /* ignore the generic itself
```

The predicate 'array_uses_element(+Freq,+Element,+DefsIn,-DefsOut,-Hard,-Soft)' creates Hard in which a new generic definition of its corresponding array is defined as using the element definition Element, and the old definition of the array.

```
 */ array_uses_element(_,Element,DefsIn,DefsIn,[],[]) :-
       read_only_local(Element),!. /*                local, with read flag
 */ array_uses_element(Freq,Element,DefsIn,DefsOut,[],Soft) :-
       map_to_ids_lexes(Element,Vars,Lexes),
       generic_def(Vars,DefsIn,GenericDef),
       GenericDef=dd([(Id,_,_)|Stars],_,_),
       unique(Id,Tag),
       (Id=global/_/_ -> N=1; N=Tag),
       NewDef=dd([(Id,N,Tag)|Stars],wr,Freq),
       drop_def(GenericDef,DefsIn,Defs1),
       drop_def(Element,Defs1,Defs2),
       ord_union([[NewDef],Defs2],DefsOut),
       sort([Element,GenericDef],Useds),
       cartesian_product([NewDef],Useds,Soft). /*          any other combination
 */ read_only_local(dd([(global/_/_,_,_)|_],_,_)) :- !,fail. /* treat global as write.
 */ read_only_local(dd(_,rd,_)) :- true.                     /* not global, read only.
```

The predicate 'drop_loop_var(+Freq,+[Id|Subs],+DefsIn,-DefsOut,-Hard,-Soft)' deletes the definitions of '[Id|Subs]' in the list 'DefsIn', yielding the list 'DefsOut'. In addition, if 'Subs' is empty, it deletes ALL definitions that have 'Id' as a subscript. This models the situation that when the subscript of an indexed variable becomes invalid, the element loses its definition. If Var has no definition, nothing much happens.

```
 */ drop_loop_var(Freq,[Sub|[]],DefsIn,DefsOut,Hard,Soft) :- !,
         drop_element_defs(Freq,Sub,DefsIn,DefsIn,Defs1,Hard,Soft),
         drop_var_def([Sub|[]],Defs1,DefsOut). /*
 */ drop_loop_var(Freq,Var,DefsIn,DefsOut,[],[]) :- !,
         drop_var_def(Var,DefsIn,DefsOut). /*
```

The predicate 'drop_element_defs(+Sub,+Defs,+DefsIn,-DefsOut,-Graph)' modifies DefsIn to give DefsOut and Graph by linking all elements that have Sub as an index to a new generic array definition.

```
 */ drop_element_defs(Freq,Sub,[Def|Defs],DefsIn,DefsOut,Hard,Soft) :-
        Def=dd([Id|Subs],_,_),member((Sub,_,_),Subs),!,
        array_uses_element(Freq,Def,DefsIn,Defs1,Hard1,Soft1),
        drop_element_defs(Freq,Sub,Defs,Defs1,DefsOut,Hard2,Soft2),
        ord_union([Hard1,Hard2],Hard),
        ord_union([Soft1,Soft2],Soft). /*
 */ drop_element_defs(Freq,Sub,[_|Defs],DefsIn,DefsOut,Hard,Soft) :-
        drop_element_defs(Freq,Sub,Defs,DefsIn,DefsOut,Hard,Soft). /*
 */ drop_element_defs(_,_,[],DefsIn,DefsIn,[],[]) :- true. /*
```

Elements are 'identical' if they have the same subscript definitions.

```
 */ link_identical_elements([],_,[],[]) :- !. /*
 */ link_identical_elements(_,[],[],[]) :- !. /*
 */ link_identical_elements([Def|Users],[Def|Useds],Hard,Soft) :-
         !,link_identical_elements(Users,Useds,Hard,Soft). /* identical defs
 */ link_identical_elements([User|Users],[Used|Useds],Hard,Soft) :-
         User=dd([(Id,_,_)|Subs],_,_),
         Used=dd([(Id,_,_)|Subs],_,_),
         !,cartesian_product([User],[Used],Join),
         ( Subs=[(*,0,0)|_] -> Soft1=Join,Hard1=[]
         ; Soft1=[],Hard1=Join
         ),
         link_identical_elements(Users,Useds,Hard2,Soft2),
         ord_union([Hard1,Hard2],Hard),
         ord_union([Soft1,Soft2],Soft). /* same index definitions
 */ link_identical_elements([User|Users],[Used|Useds],Hard,Soft) :-
         User@<Used,!,link_identical_elements(Users,[Used|Useds],Hard,Soft). /*
 */ link_identical_elements([User|Users],[Used|Useds],Hard,Soft) :-
         User@>Used,!,link_identical_elements([User|Users],Useds,Hard,Soft). /*
```

The predicate 'drop_all_locals(+Freq,+DefsIn,-DefsOut,-Hard,-Soft)' deals with procedure exit. Global variables must not be dropped, but indexed Global variables must be linked to their generic arrays.

```
 */ drop_all_locals(Freq,DefsIn,DefsOut,Hard,Soft) :-
        drop_elements(Freq,DefsIn,DefsIn,Defs1,Hard,Soft),
        retain_globals(Defs1,DefsOut). /*
```

The predicate 'drop_elements(+Freq,+Defs,+DefsIn,-DefsOut,-Hard,-Soft)' drops all array elements in the list Defs from DefsIn to give DefsOut, constructing Graph by linking each element with a 'write' flag to a new generic array definition. Simple variables and generic arrays in Defs are not dropped.

```
 */ drop_elements(Freq,[Def|Defs],DefsIn,DefsOut,Hard,Soft) :-
        drop_element(Freq,Def,DefsIn,Defs1,Hard1,Soft1),!,
        drop_elements(Freq,Defs,Defs1,DefsOut,Hard2,Soft2),
        ord_union([Soft1,Soft2],Soft),
        ord_union([Hard1,Hard2],Hard). /*
 */ drop_elements(_,[],DefsIn,DefsIn,[],[]) :- true. /*
 */ drop_element(_,dd([Id|[]],_,_),DefsIn,DefsIn,[],[]) :- !. /* simple
 */ drop_element(_,dd([Id,(*,0,0)|_],_,_),DefsIn,DefsIn,[],[]) :- !. /* generic
 */ drop_element(Freq,Def,DefsIn,DefsOut,Hard,Soft) :-
      array_uses_element(Freq,Def,DefsIn,DefsOut,Hard,Soft). /* element
```

The predicate 'link_globals_to_decls(+DefsIn,-DefsOut,-Hard,-Soft)' modifies DefsIn to produce DefsOut and Hard by linking each final global definition to an instance of its declaration definition, if it is not one already. (Declaration definitions have a static number of 1. There should be no array elements present at this time. Arrays should already have a static number of 1. Only simple globals should need modifying.)

```
 */ link_globals_to_decls([],[],[],[]) :- true. /* empty list
 */ link_globals_to_decls([Def|DefsIn],[Def|DefsOut],Hard,Soft) :-
          Def=dd([(output/_/_,_,_)],_,_),!,
          link_globals_to_decls(DefsIn,DefsOut,Hard,Soft). /* event definition
 */ link_globals_to_decls([Def|DefsIn],[Def|DefsOut],Hard,Soft) :-
          Def=dd([(_,1,_)|_],_,_),!,
          link_globals_to_decls(DefsIn,DefsOut,Hard,Soft). /* already numbered 1
 */ link_globals_to_decls([Def1|DefsIn],[Def2|DefsOut],Hard,Soft) :-
          Def1=dd([(Id,_,_)|Subs],Flag,_),
          unique(Id,Tag),Def2=dd([(Id,1,Tag)|Subs],Flag,[]),
          link_globals_to_decls(DefsIn,DefsOut,Hard,Soft1),
          ord_union([[Def2-Def1],Soft1],Soft). /* link to new definition
```

The predicate 'join_branches(+Freq,+Defs1,+Defs2,-DefsOut,-Hard,-Soft)' merges Defs1 and Defs2 to give DefsOut, in such a way that at most one definition of each variable exists in DefsOut. A 'variable' here is either a simple variable or a specific array element, ie., having particular indices and tags. Definitions common to both lists or appearing in only one list are copied to DefsOut. However, definitions of the same variable that differ are linked to a new dummy definition (numbered 0) in DefsOut. Also, if there are two definitions of the same variable, but whose index definitions differ, both definitions are linked to the generic array, and dropped.

There are two passes. First, element definitions with unequal index definitions are dropped by linking them to their generic array definitions. The predicate 'join_defs' deals with all the remaining cases, ie., simple variables, identical element definitions, definitions unique to one list only, and two definitions of the same element that have the same index definitions. This leaves the Hard cases, which is where apparently the same element appears in both lists, but it has different index definitions.

```
 */ join_branches(Freq,Defs1,Defs2,DefsOut,Hard,Soft) :-
       simplify_defs(Freq,Defs1,Defs2,Defs1,Defs3,Hard1,Soft1),
       simplify_defs(Freq,Defs2,Defs1,Defs2,Defs4,Hard2,Soft2),
       join_defs(Freq,Defs3,Defs4,DefsOut,Hard3,Soft3),
       ord_union([Hard1,Hard2,Hard3],Hard),
       ord_union([Soft1,Soft2,Soft3],Soft). /*
 */ simplify_defs(_,[],_,DefsIn,DefsIn,[],[]) :- !. /*
 */ simplify_defs(_,_,[],DefsIn,DefsIn,[],[]) :- !. /*
 */ simplify_defs(Freq,[Def1|Defs1],[Def2|Defs2],DefsIn,DefsOut,Hard,Soft) :-
        Def1=dd([(Id,_,_)|Subs],_,_),
        Def2=dd([(Id,_,_)|Subs],_,_),!,
        simplify_defs(Freq,Defs1,Defs2,DefsIn,DefsOut,Hard,Soft). /* = index defs
 */ simplify_defs(Freq,[Def1|Defs1],[Def2|Defs2],DefsIn,DefsOut,Hard,Soft) :-
        map_to_ids_lexes(Def1,Vars,_),
        map_to_ids_lexes(Def2,Vars,_),!,
        array_uses_element(Freq,Def1,DefsIn,Defs3,Hard1,Soft1),
        simplify_defs(Freq,Defs1,Defs2,Defs3,DefsOut,Hard2,Soft2),
        ord_union([Hard1,Hard2],Hard),
        ord_union([Soft1,Soft2],Soft). /* = indices but /= index definitions
 */ simplify_defs(Freq,[Def1|Defs1],[Def2|Defs2],DefsIn,DefsOut,Hard,Soft) :-
        Def1@<Def2,!,
        simplify_defs(Freq,Defs1,[Def2|Defs2],DefsIn,DefsOut,Hard,Soft). /*
 */ simplify_defs(Freq,[Def1|Defs1],[Def2|Defs2],DefsIn,DefsOut,Hard,Soft) :-
        Def1@>Def2,!,
        simplify_defs(Freq,[Def1|Defs1],Defs2,DefsIn,DefsOut,Hard,Soft). /*
 */ join_defs(Freq,[],[],[],[],[]) :- !. /* empty lists
 */ join_defs(Freq,[],[Def|Defs],[Def|DefsOut],Hard,Soft) :- !,
       join_defs(Freq,[],Defs,DefsOut,Hard,Soft). /* 1st list empty
 */ join_defs(Freq,[Def|Defs],[],[Def|DefsOut],Hard,Soft) :- !,
       join_defs(Freq,Defs,[],DefsOut,Hard,Soft). /* 2nd list empty
 */ join_defs(Freq,[Def|Defs1],[Def|Defs2],[Def|DefsOut],Hard,Soft) :-
       !,join_defs(Freq,Defs1,Defs2,DefsOut,Hard,Soft). /* same defn in both
 */ join_defs(Freq,[Def1|Defs1],[Def2|Defs2],DefsOut,Hard,Soft) :-
       Def1=dd([(Id,N1,_)|Subs],_,_),
       Def2=dd([(Id,N2,_)|Subs],_,_),!,
       (N1=:=N2 -> N=N1; N=0),
       unique(Id,Tag),NewDef=dd([(Id,N,Tag)|Subs],wr,Freq),
       sort([Def1,Def2],Useds),
       cartesian_product([NewDef],Useds,Join),
       ( Subs=[(*,0,0)|_] -> Soft1=Join,Hard1=[]
       ; Soft1=[],Hard1=Join
       ),
       join_defs(Freq,Defs1,Defs2,Defs3,Hard2,Soft2),
       ord_union([[NewDef],Defs3],DefsOut),
       ord_union([Hard1,Hard2],Hard),
       ord_union([Soft1,Soft2],Soft). /* /= defs, but = indices & = tags.
 */ join_defs(Freq,[Def1|Defs1],[Def2|Defs2],[Def1|DefsOut],Hard,Soft) :-
       Def1@<Def2,!,
       join_defs(Freq,Defs1,[Def2|Defs2],DefsOut,Hard,Soft). /* 1st only
 */ join_defs(Freq,[Def1|Defs1],[Def2|Defs2],[Def2|DefsOut],Hard,Soft) :-
       Def1@>Def2,!,
       join_defs(Freq,[Def1|Defs1],Defs2,DefsOut,Hard,Soft). /* 2nd only
SETS OF DEFINITIONS
Sets of definitions are stored as ordered lists.  Each definition has the form: dd(Vars,rd/wr,Freq), where Var is
a variable identifier in the form [Id|Subs], Lexes corresponds to the textual instance of the variable, and Tags
has the same length as Var and is a list of 'dynamic definition' (dd) identifiers.
The predicate 'find_live_def(+Var,+DefsIn,-DefOut)' succeeds if it finds the live definition 'DefOut' of variable
'Var' in the set 'DefsIn'.
 */ find_live_def([Id|Subs],Defs,Def) :-
       lookup_tags(Subs,Defs,TaggedSubs),
       Def=dd([(Id,_,_)|TaggedSubs],_,_),
       member(Def,Defs). /*
The predicate 'drop_var_def(+Var,+DefsIn,-DefsOut)' drops the live definition (if any) of variable 'Var' from the
set 'DefsIn' giving 'DefsOut'.
 */ drop_var_def(Var,[Def|DefsIn],DefsIn) :-
       map_to_ids_lexes(Def,Var,_),!. /*
 */ drop_var_def(Var,[Def|DefsIn],[Def|DefsOut]) :-
       drop_var_def(Var,DefsIn,DefsOut). /*
 */ drop_var_def(_,[],[]) :- true. /*
The predicate 'drop_def(+Var,+DefsIn,-DefsOut)' drops the live definition (if any) of variable 'Var' from the set
'DefsIn' giving 'DefsOut'.
 */ drop_def(Def,[Def|DefsIn],DefsIn) :- !. /*
 */ drop_def(Def,[Def1|DefsIn],[Def1|DefsOut]) :-
       drop_def(Def,DefsIn,DefsOut). /*
 */ drop_def(_,[],[]) :- true. /*
The predicate 'generic_def(+Var,+DefsIn,-GenericDef)' finds the definition of the array containing element Var in
the set of definitions DefsIn.
 */ generic_def([Id|Subs],DefsIn,GenericDef) :-
          dummy_list(*,Subs,Stars),
          find_live_def([Id|Stars],DefsIn,GenericDef). /*
The predicate 'live_subscript_defs(+Subs,+DefsIn,-SubDefs)' sets SubDefs to the set of all live definitions of the
simple variables in the list Subs.
 */ live_subscript_defs(Subs,DefsIn,SubDefs) :-
        live_sub_defs(Subs,DefsIn,Defs1),sort(Defs1,SubDefs). /*
 */ live_sub_defs([Sub|Subs],DefsIn,[Def|DefsOut]) :-
         find_live_def([Sub|[]],DefsIn,Def),
         live_sub_defs(Subs,DefsIn,DefsOut). /*
 */ live_sub_defs([],_,[]) :- true. /*
The predicate 'retain_globals(+DefsIn,-DefsOut)' constructs DefsOut from DefsIn by deleting all variables except
globals and event calls.
 */ retain_globals([Def|DefsIn],[Def|DefsOut]) :-
        Def=dd([(global/_/_,_,_)|_],_,_),!,
        retain_globals(DefsIn,DefsOut). /*
 */ retain_globals([Def|DefsIn],[Def|DefsOut]) :-
        Def=dd([(output/_/_,_,_)|_],_,_),!,
        retain_globals(DefsIn,DefsOut). /*
 */ retain_globals([_|DefsIn],DefsOut) :-
        retain_globals(DefsIn,DefsOut). /*
 */ retain_globals([],[]) :- true. /*
```

```
The predicate 'lookup_tags(+Subs,+Defs,-TaggedSubs)' succeeds if Subs is a list of identifiers of simple
variables, and Defs is a list of definitions, by unifying Tags with the corresponding list of the identifiers'
tags in the list of definitions.
 */ lookup_tags([],_,[]) :- !. /*
 */ lookup_tags(['*'|Ids],Defs,[(*,0,0)|TaggedSubs]) :-
          !,lookup_tags(Ids,Defs,TaggedSubs). /*
 */ lookup_tags([Id|Ids],Defs,[TaggedSub|TaggedSubs]) :-
          find_live_def([Id|[]],Defs,dd([TaggedSub],_,_)),!,
          lookup_tags(Ids,Defs,TaggedSubs). /*
 */ lookup_tags([Id|Ids],Defs,[]) :-
          write(Id),write(' has no tag definition in:'),portray(Defs). /*
End of Analyser. */
```

13.4 The Canoniser

The *Canoniser* is invoked by a goal of the form:

```
canonise(Graph,Canonical).
```

where *Graph* is the file containing the output of the *Analyser*, and *Canonical* is the file that should contain the canonical minimal process graph created by the *Canoniser*.

```
 /* The Canoniser.
The purpose of this program is to find the strongly connected components of the  use-definition graph, which
constitute the system's minimal separable components.
 */ canonise(Input,Output) :-
        read_graph(Input,Decls,FinalDefs,Hard,Soft),
        canonise_graph(Hard,Soft,SCCs,Root),
        display_sccs(Output,Decls,Hard,SCCs,Root). /*
 */ read_graph(Input,Decls,FinalDefs,Hard,Soft) :-
        seeing(User),see(Input),
        read(Decls),read(FinalDefs),read(Hard),read(Soft),
        seen,see(User). /*
The predicate 'canonise_graph(+Hard,+Soft,-SCCs,-Root)' first augments the use-definition graph GraphIn with a set
of edges that strongly connect all the dynamic definitions relating to the same static definition.  It then finds
the strong components SCCs and transitive root Root of the resulting graph.
 */ canonise_graph(Hard,Soft,SCCs,Root) :-
        'system$push$display'(message,left,'Constructing canonical graph...','','',''),
        graph_union(Hard,Soft,Graph1),
        link_static_defs(Graph1,Graph2),
        reduce(Graph2,Reduced,SCCs),
        transitive_root(Reduced,Root),
        'system$pop$display'(message). /*
 */ display_sccs(Output,Decls,Hard,SCCs,Root) :-
       vertices(Root,Vertices),
       edges(Root,Edges),
       map_sccs_to_lexes(Vertices,Edges,Lexes),
       telling(User),tell(Output),
       pretty(Decls),
       pretty(Hard),
       pretty(SCCs),
       pretty(Root),
       incidence_matrix(Lexes),
       told,tell(User). /*
The predicate 'link_static_defs(+VerticesIn,+EdgesIn,-VerticesOut,-EdgesOut)' doubly links all definitions that
correspond to the same textual instances.  It assumes that GraphIn is an ordered list of Vertex-Successors pairs.
Because Vertices have the form dd(Vars,Flag,Freq) and the list is ordered, dynamic definitions of the same textual
instance must be adjacent.
 */ link_static_defs(GraphIn,GraphOut) :-
          vertices(GraphIn,VerticesIn),
          form_double_edges(VerticesIn,Vertices1,Edges1),
          vertices_edges_to_graph(Vertices1,Edges1,Graph1),
          graph_union(GraphIn,Graph1,GraphOut). /*
The predicate 'form_double_edges(+GraphIn,-Vertices,-Edges)' sets Vertices to the Vertices of GraphIn, and Edges
to the undirected set of edges that link pairs of vertices corresponding to the same textual instances.
```

```
 */ form_double_edges([Def1,Def2|Defs],[Def1,Def2|Vs],[Def1-Def2,Def2-Def1|Edges]) :-
        Def1=dd(Vars1,_,_),Def2=dd(Vars2,_,_),
        same_lexeme(Vars1,Vars2),!,
        form_double_edges([Def2|Defs],Vs,Edges). /*
 */ form_double_edges([_|Defs],Vs,Edges) :-
        form_double_edges(Defs,Vs,Edges). /*
 */ form_double_edges([],[],[]) :- true. /*
 */ same_lexeme([],[]) :- true. /*
 */ same_lexeme([(Var,Lex,_)|Vars1],[(Var,Lex,_)|Vars2]) :- same_lexeme(Vars1,Vars2). /*
The predicate 'transitive_root(+Graph,-Root)' sets Root to the transitive root of Graph, which must be acyclic.
It does this by removing edges that correspond to composite paths.
 */ transitive_root(Graph,Root) :-
         edges(Graph,Edges),
         vertices(Graph,Vertices),
         simple_edges(Edges,Edges,Simples),
         vertices_edges_to_graph(Vertices,Simples,Root). /*
The predicate 'map_to_lexes(+GraphIn,-GraphOut)' converts a dynamic use-definition graph into its equivalent
lexical use-definition graph.
 */ map_sccs_to_lexes(Procs,Edges,Graph) :-
         map_sccs(Procs,Procs1),
         map_scc_edges(Edges,Edges1),
         vertices_edges_to_graph(Procs1,Edges1,Graph). /*
 */ map_sccs([],[]) :- true. /*
 */ map_sccs([ProcIn|ProcsIn],[ProcOut|ProcsOut]) :-
         map_scc(ProcIn,ProcOut),
         map_sccs(ProcsIn,ProcsOut). /*
 */ map_scc_edges([],[]) :- true. /*
 */ map_scc_edges([UsedIn-UserIn|EdgesIn],[UsedOut-UserOut|EdgesOut]) :-
         map_scc(UsedIn,UsedOut),
         map_scc(UserIn,UserOut),
         map_scc_edges(EdgesIn,EdgesOut). /*
 */ map_scc(DefsIn,DefsOut) :-
         map_defs(DefsIn,Defs),
         sort(Defs,DefsOut). /*
End of Canoniser. */
```

13.5 The Optimiser

The *Optimiser* is invoked by a goal of the form:

```
        optimise(Canonical,Optimal).
```

where *Canonical* is the file containing the output of the *Canoniser*, and *Optimal* is the file that should contain the optimal process graph created by the *Optimiser*.

```
/* The Optimiser.
Initially, 'SCCs' lists the initial sets of definitions (the minimal separable components), and 'classify_sccs'
assigns a degree to them to form processes.  'Root' specifies the initial graph between processes.
 */ optimise(Input,Output) :-
        read_sccs(Input,Decls,GraphIn,SCCs,Root),
        optimise_sccs(Decls,GraphIn,SCCs,Root,Procs2,GraphOut),
        display_processes(Output,Procs2,GraphOut). /*
 */ read_sccs(Input,Decls,Graph,SCCs,Root) :-
         seeing(User),see(Input),
         read(Decls),read(Graph),read(SCCs),read(Root),
         seen,see(User). /*
 */ optimise_sccs(Decls,GraphIn,SCCs,Root,Procs,GraphOut) :-
        'system$push$display'(message,left,'Optimising process graph...','','',''),
        transitive_closure(GraphIn,Closure),
        edges(Root,RootEdges),
        classify_vertices(Decls,Closure,SCCs,Procs1),
        classify_edges(Procs1,RootEdges,Edges1),
        length(Procs1,N),
        'system$show$progress',
        improve_solution(0,N,Decls,Closure,Procs1,Edges1,Procs2,Edges2),
        'system$hide$progress',
        vertices_edges_to_graph(Procs2,Edges2,Graph1),
        map_to_lexes(Graph1,GraphOut),
        top_sort(GraphOut,Procs),
        'system$pop$display'(message). /*
 */ display_processes(Output,Procs2,GraphOut) :-
         telling(User),tell(Output),
         pretty(Procs2),pretty(GraphOut),
         incidence_matrix(GraphOut),
         told,tell(User). /*
```

The predicate 'classify_sccs(+Decls,+Graph,+SCCs,-Procs)' succeeds by associating with each SCC in the list SCCs a 'Degree', which is a measure of the parallelism possible within the SCC.

```
 */ classify_vertices(Decls,Graph,[SCC|SCCs],[(Degree,SCC)|Procs]) :-
        classify_process(Decls,Graph,SCC,Degree),
        classify_vertices(Decls,Graph,SCCs,Procs). /*
 */ classify_vertices(_,_,[],[]) :- true. /*
```

The predicate 'classify_process(+Decls,+Graph,+Defs,-Degree)' succeeds by associating with Defs a 'Degree', which is a measure of the parallelism possible within it.

```
 */ classify_process(Decls,Graph,Defs,Degree) :-
      sort(Defs,Defs1),
      subgraph(Graph,Defs1,SubGraph1),
      invalid_loops(SubGraph1,Invalid),
      adjust_graph_freqs(Defs1,SubGraph1,SubGraph),
      classify_users(Decls,Invalid,SubGraph,[*],Degree). /*
```

The predicate 'adjust_graph_freqs(+Defs,+GraphIn,-GraphOut)' succeeds by changing the frequencies in GraphIn to give GraphOut in such a way that for each definition the calling frequency is eliminated.

```
 */ adjust_graph_freqs(Defs,GraphIn,GraphOut) :-
         vertex_freqs(Defs,GraphIn,VertexFreqs),
         adjust_graph(VertexFreqs,VertexFreqs,GraphIn,GraphOut). /*
 */ vertex_freqs(_,[],[]) :- true. /*
 */ vertex_freqs(Defs,[VertexIn-_|Graph],[VertexOut|Vertices]) :-
         VertexIn=dd(Var,Flag,FreqIn),
         remove_call_freq(Defs,FreqIn,FreqOut),
         VertexOut=dd(Var,Flag,FreqOut),
         vertex_freqs(Defs,Graph,Vertices). /*
 */ remove_call_freq(_,[],[]) :- true. /*
 */ remove_call_freq(Defs,VarsIn,VarsIn) :-
         VarsIn=[(Id,Lexes,_)|_],
         member(dd([(Id,Lexes,_)],_,_),Defs),!. /* Var IS defined in this process
 */ remove_call_freq(Defs,[_|VarsIn],VarsOut) :-
         remove_call_freq(Defs,VarsIn,VarsOut). /* Var is NOT defined in this process
```

The predicate 'adjust_graph(+VertexFreqs,+VertexFreqs,+GraphIn,-GraphOut)' is given a list VertexFreqs of vertices whose frequencies have been adjusted. This list should be 1:1 with the list of vertices in the graph. So it is just a matter of replacing corresponding vertices, then fixing the edges.

```
 */ adjust_graph(_,_,[],[]) :- true. /*
 */ adjust_graph(VertexFreqs,[Vertex1|Vertices1],[Vertex2-Edges1|GraphIn],
                     [Vertex1-Edges2|GraphOut]) :-
      Vertex1=dd(Vars,_,_),Vertex2=dd(Vars,_,_),!,
      adjust_edges(VertexFreqs,Edges1,Edges2),
      adjust_graph(VertexFreqs,Vertices1,GraphIn,GraphOut). /*
 */ adjust_graph(VertexFreqs,[_|Vertices],GraphIn,GraphOut) :-
      adjust_graph(VertexFreqs,Vertices,GraphIn,GraphOut). /*
 */ adjust_edges(_,[],[]) :- true. /*
 */ adjust_edges([Vertex1|Vertices1],[Vertex2|Vertices2],[Vertex1|Vertices3]) :-
      Vertex1=dd(Vars,_,_),Vertex2=dd(Vars,_,_),!,
      adjust_edges(Vertices1,Vertices2,Vertices3). /*
 */ adjust_edges([_|Vertices1],Vertices2,Vertices3) :-
      adjust_edges(Vertices1,Vertices2,Vertices3). /*
```

The predicate 'invalid_loops(+Defs,-Invalid)' builds a list of invalid loop variables, ie., those that appear non-hierarchically. Note that sorting is used to remove duplicates.

```
 */ invalid_loops(Graph,Invalid) :-
      frequencies(Graph,Freqs),
      sort(Freqs,Loops),
      def_def_conflicts(Loops,Loops,[],Known),
      sort(Known,Invalid). /*
```

Accumulate all frequencies.

```
 */ frequencies([],[]) :- true. /*
 */ frequencies([dd(_,_,Freq1)-_|Graph],[Freq2|Freqs]) :-
         strip_tags(Freq1,Freq2),
         frequencies(Graph,Freqs). /*
 */ strip_tags([],[]) :- true. /*
 */ strip_tags([(Id,Lex,_)|Freq1],[(Id,Lex,0)|Freq2]) :-
         strip_tags(Freq1,Freq2). /*
```

Consider all frequency pairs.

```
 */ def_def_conflicts(_,[],Invalid,Invalid) :- true. /*
 */ def_def_conflicts(Freqs1,[Freq|Freqs2],InvalidIn,InvalidOut) :-
      def_conflicts(Freqs1,Freq,[],Invalid1),!,
      append(Invalid1,InvalidIn,Invalid2),
      def_def_conflicts(Freqs1,Freqs2,Invalid2,InvalidOut). /*
Match a frequency against those for the same event.
 */ def_conflicts([],_,Invalid,Invalid) :- true. /*
 */ def_conflicts([Freq1|Freqs],Freq2,InvalidIn,InvalidOut) :-
      Freq1=[(_/Event/_,_,_)|_],
      Freq2=[(_/Event/_,_,_)|_],!,
      freq_freq_conflicts(Freq1,Freq2,Freq1,InvalidIn,Invalid1),
      def_conflicts(Freqs,Freq2,Invalid1,InvalidOut). /*
 */ def_conflicts([_|Freqs],Freq2,InvalidIn,InvalidOut) :-
      def_conflicts(Freqs,Freq2,InvalidIn,InvalidOut). /*
Check each term of a frequency against the other, rejecting all terms following a while loop.
 */ freq_freq_conflicts(_,_,[],Invalid,Invalid) :- true. /*
 */ freq_freq_conflicts(Freq1,Freq2,[Var|Vars],InvalidIn,InvalidOut) :-
       var_freq_conflicts(Freq1,Freq2,Var,Freq2,InvalidIn,Invalid1),
       freq_freq_conflicts(Freq1,Freq2,Vars,Invalid1,InvalidOut). /*
    freq_freq_conflicts(Freq1,Freq2,Freq,InvalidIn,InvalidOut) :-
       Freq=[Var|Vars],
       ( Var=(loop/_/_,_,_) ->
           var_freq_conflicts(Freq1,Freq2,Var,Freq2,InvalidIn,Invalid1),
           freq_freq_conflicts(Freq1,Freq2,Vars,Invalid1,InvalidOut)
       ; Var=(internal/_/_,_,_) ->
           append(Vars,InvalidIn,InvalidOut)
       ).
Check one term against all terms of the other frequency, rejecting all terms following a while loop.
 */ var_freq_conflicts(_,_,_,[],Invalid,Invalid) :- true. /*
 */ var_freq_conflicts(Freq1,Freq2,Var1,[Var2|Vars],InvalidIn,InvalidOut) :-
       var_var_conflicts(Freq1,Freq2,Var1,Var2,InvalidIn,Invalid1),
       var_freq_conflicts(Freq1,Freq2,Var1,Vars,Invalid1,InvalidOut). /*
    var_freq_conflicts(Freq1,Freq2,Var1,Freq,InvalidIn,InvalidOut) :-
       Freq=[Var2|Vars],
       ( Var2=(loop/_/_,_,_) ->
           var_var_conflicts(Freq1,Freq2,Var1,Var2,InvalidIn,Invalid1),
           var_freq_conflicts(Freq1,Freq2,Var1,Vars,Invalid1,InvalidOut)
       ; Var2=(internal/_/_,_,_) ->
           append(Vars,InvalidIn,InvalidOut)
       ).
Check if two terms are hierarchically related.  Reject both if not.
 */ var_var_conflicts(Freq1,Freq2,Var1,Var2,InvalidIn,InvalidOut) :-
      ( ( member(Var1,Freq2)
        ; member(Var2,Freq1)
        ) -> InvalidOut=InvalidIn
      ; true -> InvalidOut=[Var1,Var2|InvalidIn]
      ). /*
The predicate 'classify_users(+Decls,+Closure,-Degree)' sets Degree to the common degree of parallelism possible
for the definitions that are the vertices of Closure.  In Closure, each definition is paired with a list of all
the definitions from which it is derived.  The degree of Closure is the common prefix of each definition in it.
 */ classify_users(_,_,[],Degree,Degree). /* done
 */ classify_users(_,_,_,[],[]) :- !. /*    pointless to continue
 */ classify_users(Decls,Invalid,[User-Useds|Uses],Degree0,Degree) :-
        classify_user(Decls,Invalid,User,Useds,Degree0,Degree1),
        classify_users(Decls,Invalid,Uses,Degree1,Degree). /*
The predicate 'classify_user(+Decls,+Invalid,+User,Useds,-Degree)' computes the degree of each definition by
finding the common prefix of the indices of the definition and all its antecedents, then converting this to a list
of codomains, or data types.  After this,  the degree of parallelism is checked against the loop structure, and
reduced if necessary.
 */ classify_user(Decls,Invalid,User,Useds,DomsIn,DomsOut) :-
      classify_def(User,Invalid,Indices1,_),
      codomains(Decls,Indices1,Doms1),
      classify_useds(Decls,Invalid,Useds,Indices1,Doms1,Doms2),
      common_codomains(DomsIn,Doms2,DomsOut). /*
The predicate 'classify_useds(+Decls,+Invalid,+Useds,+Indices,-Degree,-Rate)' finds the common prefix of the
degree (Degree) and of the rate (Rate) of each definition in the set Useds.
 */ classify_useds(_,_,[],_,Doms,Doms) :- true. /* done
 */ classify_useds(_,_,_,[],Doms,Doms) :- !. /*    pointless to continue
 */ classify_useds(Decls,Invalid,[Used|Useds],IndicesIn,DomsIn,DomsOut) :-
        classify_def(Used,Invalid,Indices1,_),
        common_indices(IndicesIn,Indices1,Indices2),
        codomains(Decls,Indices2,Doms1),
        common_codomains(DomsIn,Doms1,Doms2),
        classify_useds(Decls,Invalid,Useds,IndicesIn,Doms2,DomsOut). /*
```

The predicate 'classify_def(+Def,-Indices,-Rate)' sets Indices to the simplest list of loop variable definitions that could enclose Def, terminated with [*] and [0] if it is 'extensible', i.e., inner loops are allowed. A state variable element allows parallelism by its subscripts. A generic array reference is treated as extensible ([*]). Local variables are also extensible.

```
 */ classify_def(dd([(global/_/_,_,_)|Subs],_,Freq),Invalid,Indices,Freq) :- !,
        valid_prefix(Invalid,Subs,Indices). /* state variable (can't be a loop variable)
 */ classify_def(dd([_|Subs],_,Freq),Invalid,Indices,Freq) :-
        find_degree(Subs,Freq,Degree),
        valid_prefix(Invalid,Degree,Degree1),
        ( Degree=Degree1 -> append(Subs,[(*,0,0)],Indices)
        ; true ->           Indices=Degree1
        ). /* local variable (may be an invalid loop variable, or have invalid indices)
 */ find_degree(Subs,FreqIn,Degree) :-
        remove_subs_from_freq(Subs,FreqIn,FreqOut),
        append(Subs,FreqOut,Degree). /*
 */ remove_subs_from_freq(_,[],[]) :- true. /*
 */ remove_subs_from_freq(Subs,[(Var,_,_)|FreqIn],FreqOut) :-
        member((Var,_,_),Subs),!,
        remove_subs_from_freq(Subs,FreqIn,FreqOut). /*
 */ remove_subs_from_freq(Subs,[Var|FreqIn],[Var|FreqOut]) :-
        remove_subs_from_freq(Subs,FreqIn,FreqOut). /*
 */ valid_prefix(_,[],[]) :- true. /* done
 */ valid_prefix(Invalid,[(internal/_/_,_,_)|_],[]) :- !. /*        while loop variable
 */ valid_prefix(Invalid,[(Id,_,_)|_],[]) :- member((Id,_,_),Invalid),!. /* bad loop var
 */ valid_prefix(Invalid,[Sub|Subs],[Sub|Indices]) :-
        valid_prefix(Invalid,Subs,Indices). /*      OK so far.
```

The predicate 'common_indices(+Subs1,+Subs2,-Prefix)' succeeds if Subs1 and Subs2 are lists of tagged subscripts and Prefix is their longest common prefix. A subscript of [*] is a wild card and matches anything.

```
 */ common_indices([(*,0,0)|_],Indices,Indices) :- !. /*
 */ common_indices(Indices,[(*,0,0)|_],Indices) :- !. /*
 */ common_indices([Index|Indices1],[Index|Indices2],[Index|Indices3]) :-
         !,common_indices(Indices1,Indices2,Indices3). /*
 */ common_indices(_,_,[]) :- true. /*
```

The predicate 'common_codomains(+Doms1,+Doms2,-Prefix)' succeeds if Doms1 and Doms2 are lists of domains and Prefix is their longest common prefix. A subscript of '*' is a wild card and matches anything.

```
 */ common_codomains([*],Doms,Doms) :- !. /*
 */ common_codomains(Doms,[*],Doms) :- !. /*
 */ common_codomains([Dom|Doms1],[Dom|Doms2],[Dom|Doms3]) :-
          !,common_codomains(Doms1,Doms2,Doms3). /*
 */ common_codomains(_,_,[]) :- true. /*
```

The predicate 'codomains(+Decls,+Prefix,-Codoms)' succeeds by finding the declaration of each term of Prefix in Decls, and setting the terms of Codoms to their corresponding codomains, i.e., types.

```
 */ codomains(_,[],[]) :- !. /*
 */ codomains(Decls,[(*,0,0)|Subs],[*|Codoms]) :-
         !,codomains(Decls,Subs,Codoms). /*
 */ codomains(Decls,([(Id,_,_)|Subs]),[Codom|Codoms]) :-
         find_declaration(Id,Decls,(_,Codom)),
         codomains(Decls,Subs,Codoms). /*
```

The predicate 'find_declaration(+Id,+Decls,-Decl)' succeeds if Decls is a list of declarations, and Decl is the (unique) declaration of the identifier Id.

```
 */ find_declaration(Id,[],(lex([Id],1),'*')) :- !,
          write('No declaration for '),writeq(Id),nl. /*
 */ find_declaration(Id,[Decl|Decls],Decl) :- Decl=(lex([Id|_],_),_),!. /*
 */ find_declaration(Id,[_|Decls],Decl) :- find_declaration(Id,Decls,Decl),!. /*
 */ find_declaration(Id,Decls,(lex([Id],1),'*')) :- !,
          write('No declaration for '),writeq(Id),portray(Decls). /*
```

The predicate 'classify_edges(+Procs,+EdgesIn,-EdgesOut)' extends EdgesIn to Edges out by giving each process its degree from Procs.

```
 */ classify_edges(Procs,[User-Used|Defs],[Proc1-Proc2|Edges]) :-
          look_up_process(User,Procs,Proc1),
          look_up_process(Used,Procs,Proc2),
          classify_edges(Procs,Defs,Edges). /*
 */ classify_edges(_,[],[]). /*
```

The predicate 'improve_solution(+Decls,+Graph,+ProcsIn,+EdgesIn, -ProcsOut,-EdgesOut)' succeeds by improving the components specified by ProcsIn, with Edges EdgesIn, to derive ProcsOut and EdgesOut, by repeatedly first composing all possible pairs of neighbours in EdgesIn, then a pair of processes that are not connected in EdgesIn, until no more compositions are possible.

```
 */ improve_solution(M,N,Decls,Graph,ProcsIn,EdgesIn,ProcsOut,EdgesOut) :-
         'system$set$progress'(M,N),M1 is M+1,
         attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1),
         !,simple_edges(Edges1,Edges1,Edges2),
         improve_solution(M1,N,Decls,Graph,Procs1,Edges2,ProcsOut,EdgesOut). /*
 */ improve_solution(_,_,_,_,ProcsIn,EdgesIn,ProcsIn,EdgesIn). /*
```

The order of the following clauses affects the priority of applying hauristics.

```
 */ attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1) :-
        merge_up(state,Decls,Graph,EdgesIn,ProcsIn,EdgesIn,Procs1,Edges1),!. /*
 */ attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1) :-
        merge_down(state,Decls,Graph,EdgesIn,ProcsIn,EdgesIn,Procs1,Edges1),!. /*
 */ attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1) :-
        state_procs(ProcsIn,StateProcs),
        merge_disjoint(state,Decls,Graph,(StateProcs,StateProcs,StateProcs),
                       ProcsIn,EdgesIn,Procs1,Edges1),!. /*
 */ attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1) :-
        merge_up(mixed,Decls,Graph,EdgesIn,ProcsIn,EdgesIn,Procs1,Edges1),!. /*
 */ attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1) :-
        merge_down(mixed,Decls,Graph,EdgesIn,ProcsIn,EdgesIn,Procs1,Edges1),!. /*
 */ attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1) :-
        state_procs(ProcsIn,StateProcs),
        local_procs(ProcsIn,LocalProcs),
        merge_disjoint(mixed,Decls,Graph,(StateProcs,LocalProcs,LocalProcs),
                       ProcsIn,EdgesIn,Procs1,Edges1),!. /*
 */ attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1) :-
        merge_up(local,Decls,Graph,EdgesIn,ProcsIn,EdgesIn,Procs1,Edges1),!. /*
 */ attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1) :-
        merge_down(local,Decls,Graph,EdgesIn,ProcsIn,EdgesIn,Procs1,Edges1),!. /*
 */ attempt_improvement(Decls,Graph,ProcsIn,EdgesIn,Procs1,Edges1) :-
        local_procs(ProcsIn,LocalProcs),
        merge_disjoint(local,Decls,Graph,(LocalProcs,LocalProcs,LocalProcs),
                       ProcsIn,EdgesIn,Procs1,Edges1),!. /*
```

The predicate 'merge_up(+Mode,+Decls,+Graph,+Candidates,+ProcsIn,+EdgesIn, -ProcsOut,-EdgesOut)' succeeds if Candidates contains an edge between two compatible processes, deriving ProcOut by replacing them by their composition, where the composition replaces the used Proc, and the used proc is a state variable.

```
 */ merge_up(Mode,Decls,Graph,[(User-Used)|_],ProcsIn,EdgesIn,ProcsOut,EdgesOut) :-
       User=(Degree1,Defs1),Used=(Degree2,_),
       ( Mode=state -> non_extensible(Degree1),non_extensible(Degree2)
       ; Mode=mixed -> extensible(Degree1),non_extensible(Degree2)
       ; Mode=local -> extensible(Degree1),extensible(Degree2)
       ),
       not_bare_loop_variable(Defs1),
       merge_processes(up,Mode,Decls,Graph,User,Used,ProcsIn,EdgesIn,
                  ProcsOut,EdgesOut),!. /*
 */ merge_up(Mode,Decls,Graph,[Proc1-Proc2|Rest],ProcsIn,EdgesIn,ProcsOut,EdgesOut) :-
       merge_up(Mode,Decls,Graph,Rest,ProcsIn,EdgesIn,ProcsOut,EdgesOut). /*
 */ not_bare_loop_variable(Used) :- bare_loop_variable(Used),!,fail. /*
 */ not_bare_loop_variable(_) :- true. /*
 */ bare_loop_variable([dd([(loop/Event/Id,_,_)],_,_)|Defs]) :-
       defs_match_loop(loop/Event/Id,Defs). /*
 */ defs_match_loop(_,[]) :- true. /*
 */ defs_match_loop(Var,[dd([(Var,_,_)],_,_)|Defs]) :-
       defs_match_loop(Var,Defs). /*
```

The predicate 'merge_down(+Mode,+Decls,+Graph,+Candidates,+ProcsIn,+EdgesIn, -ProcsOut,-EdgesOut)' succeeds if Candidates contains an edge between two compatible processes, deriving ProcOut by replacing them by their composition, where the composition replaces the 'upstream' Proc.

```
 */ merge_down(Mode,Decls,Graph,[(User-Used)|_],ProcsIn,EdgesIn,ProcsOut,EdgesOut):-
       User=(Degree1,_),Used=(Degree2,_),
       ( Mode=state -> non_extensible(Degree1),non_extensible(Degree2)
       ; Mode=mixed -> non_extensible(Degree1),extensible(Degree2)
       ; Mode=local -> extensible(Degree1),extensible(Degree2)
       ),
       merge_processes(down,Mode,Decls,Graph,Used,User,ProcsIn,EdgesIn,
                  ProcsOut,EdgesOut),!. /*
 */ merge_down(Mode,Decls,Graph,[Proc1-Proc2|Rest],ProcsIn,EdgesIn,ProcsOut,EdgesOut) :-
       merge_down(Mode,Decls,Graph,Rest,ProcsIn,EdgesIn,ProcsOut,EdgesOut). /*
```

Called by merge_up or merge_down, with different parameters.

```
 */ merge_processes(Dir,Mode,Decls,Graph,User,Used,ProcsIn,EdgesIn,ProcsOut,EdgesOut) :-
       up_compatible(Decls,Graph,Used,User,Process3),!,
       delete_process(ProcsIn,User,ProcsIn1),
       replace_process(ProcsIn1,Used,Process3,ProcsOut),
       merge_edges(Dir,Mode,User,Used,Process3,EdgesIn,EdgesOut). /*
```

The predicate 'merge_disjoint(+Mode,+Decls,+Graph,+Procs1,+Procs2,+EdgesIn, -ProcsOut,-EdgesOut)' succeeds if a pair of processes (unordered) drawn from the lists Proc1 and Proc2 are not connected but compatible, deriving ProcOut by replacing them by their composition.

```
 */ merge_disjoint(_,_,_,([],[],_),_,_,_,_) :- !,fail. /* all done
 */ merge_disjoint(Mode,Decls,Graph,([_|Procs1],[],Procs2),ProcsIn,EdgesIn,
                ProcsOut,EdgesOut) :- !,
        merge_disjoint(Mode,Decls,Graph,(Procs1,Procs2,Procs2),
                    ProcsIn,EdgesIn,ProcsOut,EdgesOut). /* 2nd list exhausted
 */ merge_disjoint(Mode,Decls,Graph,([Proc|Procs1],[Proc|Procs2],Procs3),
                ProcsIn,EdgesIn,ProcsOut,EdgesOut) :- !,
        merge_disjoint(Mode,Decls,Graph,([Proc|Procs1],Procs2,Procs3),
                    ProcsIn,EdgesIn,ProcsOut,EdgesOut). /* not a pair
 */ merge_disjoint(Mode,Decls,Graph,([Proc1|Procs1],[Proc2|Procs2],Procs3),
                ProcsIn,EdgesIn,ProcsOut,EdgesOut) :-
        descendant(Proc1,EdgesIn,Proc2),!,
        merge_disjoint(Mode,Decls,Graph,([Proc1|Procs1],Procs2,Procs3),
                    ProcsIn,EdgesIn,ProcsOut,EdgesOut). /* Proc1 --> Proc2
 */ merge_disjoint(Mode,Decls,Graph,([Proc1|Procs1],[Proc2|Procs2],Procs3),
                ProcsIn,EdgesIn,ProcsOut,EdgesOut) :-
        descendant(Proc2,EdgesIn,Proc1),!,
        merge_disjoint(Mode,Decls,Graph,([Proc1|Procs1],Procs2,Procs3),
                    ProcsIn,EdgesIn,ProcsOut,EdgesOut). /* Proc2 --> Proc1
 */ merge_disjoint(Mode,Decls,Graph,([Proc1|Procs1],[Proc2|Procs2],Procs3),
                ProcsIn,EdgesIn,ProcsOut,EdgesOut) :-
        compatible(Decls,Graph,Proc1,Proc2,Proc3),!,
        delete_process(ProcsIn,Proc1,ProcsIn1),
        replace_process(ProcsIn1,Proc2,Proc3,ProcsOut),
        merge_edges(disjoint,Mode,Proc1,Proc2,Proc3,EdgesIn,EdgesOut). /* compatible
 */ merge_disjoint(Mode,Decls,Graph,(Procs1,[_|Procs2],Procs3),
                ProcsIn,EdgesIn,ProcsOut,EdgesOut) :- !,
        merge_disjoint(Mode,Decls,Graph,(Procs1,Procs2,Procs3),
                    ProcsIn,EdgesIn,ProcsOut,EdgesOut). /* incompatible
```

The predicate 'merge_edges(+Proc1,+Proc2,+Proc3,+EdgesIn,-EdgesOut)' succeeds by redirecting edges in EdgesIn to give EdgesOut so that any edge that joined Proc1 or Proc2 now joins Proc3. The resulting set of edges is then sorted to remove duplicates.

```
 */ merge_edges(Dir,Mode,Proc1,Proc2,Proc3,EdgesIn,EdgesOut) :-
      adjust_edges(Proc1,Proc2,Proc3,EdgesIn,Edges1),
      trace_merge(Dir,Mode,Proc1,Proc2,Proc3),
      sort(Edges1,EdgesOut). /*
```

The predicate 'trace_merge(+Mode,+Proc1,+Proc2)' traces merges taking place during optimisation. Place the rule for the desired level of tracing first!

```
*/ trace_merge(_,_,_,_,_) :- !.                    /* Tracing OFF, if placed 1st.
*/ trace_merge(Dir,Mode,Proc1,Proc2,Proc3) :- !,
        portray(merge(Dir,Mode,Proc1,Proc2,Proc3)).       /* Detailed tracing, if 1st.
*/ trace_merge(Dir,Mode,Proc1,Proc2,Proc3) :- !,
        map_process(Proc1,Lexes1),
        map_process(Proc2,Lexes2),
        map_process(Proc3,Lexes3),
        portray(merge(Dir,Mode,Lexes1,Lexes2,Lexes3)).  /* Definition Tracing, if 1st.
*/ trace_merge(Dir,Mode,Proc1,Proc2,Proc3) :- !,
        map_to_names(Proc1,Lexes1),
        map_to_names(Proc2,Lexes2),
        map_to_names(Proc3,Lexes3),
        write('merge('),write(Dir),write(','),write(Mode),write(','),nl,
        write('      '),write(Lexes1),write(','),nl,
        write('      '),write(Lexes2),write(','),nl,
        write('      '),write(Lexes3),write(').'),nl.  /* Name Tracing, if placed 1st.
*/ trace_merge(Dir,Mode,(Degree1,_),(Degree2,_),(Degree3,_)) :- !,
        write(merge(Dir,Mode,Degree1,Degree2,Degree3)),nl. /* Degree trace on, if 1st.
```

The predicate 'adjust_edges(+Proc1,+Proc2,+Proc3,+EdgesIn,-EdgesOut)' succeeds by redirecting edges in EdgesIn to give EdgesOut so that any edge that joined Proc1 or Proc2 now joins Proc3. The resulting set of edges may include duplicates. The following special cases need to be considered: 1) The edge joins Procs 1 and 2; 2) The edge enters Proc 1 or 2; 3) The edge leaves Proc 1 or 2; 4) Any other edge.

```
 */ adjust_edges(Proc1,Proc2,Proc3,[(Proc1-Proc2)|EdgesIn],EdgesOut) :- !,
         adjust_edges(Proc1,Proc2,Proc3,EdgesIn,EdgesOut). /*
 */ adjust_edges(Proc1,Proc2,Proc3,[(Proc2-Proc1)|EdgesIn],EdgesOut) :- !,
         adjust_edges(Proc1,Proc2,Proc3,EdgesIn,EdgesOut). /*
 */ adjust_edges(Proc1,Proc2,Proc3,[(Proc1-Proc4)|EdgesIn],[Proc3-Proc4|EdgesOut]) :-
         !,adjust_edges(Proc1,Proc2,Proc3,EdgesIn,EdgesOut). /*
 */ adjust_edges(Proc1,Proc2,Proc3,[(Proc2-Proc4)|EdgesIn],[Proc3-Proc4|EdgesOut]) :-
         !,adjust_edges(Proc1,Proc2,Proc3,EdgesIn,EdgesOut). /*
 */ adjust_edges(Proc1,Proc2,Proc3,[(Proc4-Proc1)|EdgesIn],[Proc4-Proc3|EdgesOut]) :-
         !,adjust_edges(Proc1,Proc2,Proc3,EdgesIn,EdgesOut). /*
 */ adjust_edges(Proc1,Proc2,Proc3,[(Proc4-Proc2)|EdgesIn],[Proc4-Proc3|EdgesOut]) :-
         !,adjust_edges(Proc1,Proc2,Proc3,EdgesIn,EdgesOut). /*
 */ adjust_edges(Proc1,Proc2,Proc3,[Edge|EdgesIn],[Edge|EdgesOut]) :-
         adjust_edges(Proc1,Proc2,Proc3,EdgesIn,EdgesOut). /*
 */ adjust_edges(_,_,_,[],[]) :- true. /*
```

The predicate 'delete_process(+ProcsIn,+Proc,-ProcsOut)' succeeds if ProcsOut is derived from ProcsIn by removing all instances of Proc.

```
 */ delete_process([],_,[]) :- true. /*
 */ delete_process([Proc|ProcsIn],Proc,ProcsOut) :-
        !,delete_process(ProcsIn,Proc,ProcsOut). /*
 */ delete_process([Proc1|ProcsIn],Proc,[Proc1|ProcsOut]) :-
        delete_process(ProcsIn,Proc,ProcsOut). /*
```

The predicate 'replace_process(+ProcsIn,+Proc1,+Proc2,-ProcsOut)' succeeds if ProcsOut is derived from ProcsIn by replaing all instances of Proc1 by Proc2.

```
 */ replace_process([],_,_,[]) :- true. /*
 */ replace_process([Proc1|ProcsIn],Proc1,Proc2,[Proc2|ProcsOut]) :-
        !,replace_process(ProcsIn,Proc1,Proc2,ProcsOut). /*
 */ replace_process([Proc|ProcsIn],Proc1,Proc2,[Proc|ProcsOut]) :-
        replace_process(ProcsIn,Proc1,Proc2,ProcsOut). /*
```

The predicate 'compatible(+Decls,+Graph,+Proc1,+Proc2,-ProcOut)' succeeds if Proc1 and Proc2 can be combined without loss of parallelism, setting ProcOut to the composition of Proc1 and Proc2. The 1st rule is defensive programming.

```
 */ compatible(_,_,(_,Defs),(_,Defs),_) :- !,portray(compatible_error(Defs)),fail. /*
 */ compatible(Decls,Graph,Proc1,Proc2,ProcOut) :-
        up_compatible(Decls,Graph,Proc1,Proc2,ProcOut),!. /*
 */ compatible(Decls,Graph,Proc1,Proc2,ProcOut) :-
        up_compatible(Decls,Graph,Proc2,Proc1,ProcOut). /*
 */ up_compatible(Decls,Graph,(Degree1,Defs1),(Degree2,Defs2),ProcOut) :-
        subsumes(Decls,Graph,(Degree1,Defs1),(Degree2,Defs2),ProcOut),!. /*
 */ up_compatible(Decls,Graph,(Degree1,Defs1),(Degree2,Defs2),ProcOut) :-
        promotable(Decls,Graph,(Degree2,Defs2),(Degree1,Defs1),ProcOut),!. /*
```

The predicate 'promotable(+Proc1,+Proc2,-ProcOut)' succeeds if the degree of Proc1 can be extended to the degree of Proc2.

```
 */ promotable(Decls,Graph,(Degree1,Defs1),(Degree2,Defs2),(DegreeOut,DefsOut)) :-
         extends_to(Degree1,Degree2),!,
         append(Defs1,Defs2,DefsOut),
         classify_process(Decls,Graph,DefsOut,DegreeOut),!,
         no_worse_degree(DegreeOut,Degree2). /*
 */ extends_to([*],[]) :- !. /*
 */ extends_to([*],[Dom|Doms]) :- !,extends_to([*],Doms). /*
 */ extends_to([],[]) :- !. /*
 */ extends_to([Dom|Doms1],[Dom|Doms2]) :- extends_to(Doms1,Doms2). /*
 */ state_procs([(Degree,Defs)|ProcsIn],[(Degree,Defs)|ProcsOut]) :-
        non_extensible(Degree),!,
        state_procs(ProcsIn,ProcsOut). /*
 */ state_procs([_|ProcsIn],ProcsOut) :- !,
        state_procs(ProcsIn,ProcsOut). /*
 */ state_procs([],[]). /*
 */ local_procs([(Degree,_)|ProcsIn],ProcsOut) :-
        non_extensible(Degree),!,
        local_procs(ProcsIn,ProcsOut). /*
 */ local_procs([Proc|ProcsIn],[Proc|ProcsOut]) :- !,
        local_procs(ProcsIn,ProcsOut). /*
 */ local_procs([],[]). /*
 */ extensible([*]) :- !. /*
 */ extensible([_|T]) :- extensible(T). /*
 */ non_extensible(X) :- extensible(X),!,fail. /*
 */ non_extensible(_). /*
```

The predicate 'subsumes(+Proc1,+Proc2,-ProcOut)' succeeds if Proc1 has less parallelism than Proc2, but Proc2 accesses only indices already accessed by Proc1, so that, although the resulting parallelism is only that of Proc1, Proc2 is accessed 'free'; all provided that their composition has parallelism as great as that of Proc1.

```
 */ subsumes(Decls,Graph,(Degree1,Defs1),(Degree2,Defs2),(DegreeOut,DefsOut)) :-
         less_or_equal_degree(Degree1,Degree2),
         extract_subscripts(Defs1,Subs1),sort(Subs1,Set1),
         extract_subscripts(Defs2,Subs2),sort(Subs2,Set2),
         ord_subset(Set2,Set1),!,
         append(Defs1,Defs2,DefsOut),
         classify_process(Decls,Graph,DefsOut,DegreeOut),!,
         no_worse_degree(DegreeOut,Degree1). /*
```

The predicate 'extract_subscripts(+Defs,-SubsList)' collects all the subscripts used by the set of definitions Defs in the list SubsList, with the exception of simple local variables.

```
 */ extract_subscripts([dd([(global/_/_,_,_)|Subs1],_,_)|Defs],[Subs2|SubsList]) :-
        !,simplify_to_lexes(Subs1,Subs2),
        extract_subscripts(Defs,SubsList). /*
 */ extract_subscripts([dd([_|[]],_,_)|Defs],SubsList) :-
        !,extract_subscripts(Defs,SubsList). /*
 */ extract_subscripts([dd([_|Subs1],_,_)|Defs],[Subs2|SubsList]) :-
        simplify_to_lexes(Subs1,Subs2),
        extract_subscripts(Defs,SubsList). /*
 */ extract_subscripts([],[]) :- true. /*
 */ simplify_to_lexes([],[]) :- true. /*
 */ simplify_to_lexes([(Id,Lex,_)|Subs1],[(Id,Lex)|Subs2]) :-
        simplify_to_lexes(Subs1,Subs2). /*
The predicate 'less_or_equal_degree(+X,+Y)' succeeds if X is a leading subsequence of Y, ignoring wild cards.
 */ less_or_equal_degree([],_) :- true. /*
 */ less_or_equal_degree([*],_) :- true. /*
 */ less_or_equal_degree([H|T1],[H|T2]) :- less_or_equal_degree(T1,T2). /*
The predicate 'look_up_process(+Defs,+Procs,-Proc)' succeeds if Proc is the process containing Defs in the list
Procs.
 */ look_up_process(Defs,[(Degree,Defs)|_],(Degree,Defs)) :- !. /*
 */ look_up_process(Defs,[_|Procs],Proc) :-
          look_up_process(Defs,Procs,Proc). /*
The predicate 'no_worse_degree(+Degree1,+Degree2)' succeeds if Degree1 and Degree2 are the same or if one is the
same as the other extended by '*', OR if they agree to the point where one has an *, eg, [a,*],[a].
 */ no_worse_degree([],[]) :- true. /*
 */ no_worse_degree(_,[*]) :- true. /*
 */ no_worse_degree([*],_) :- true. /*
 */ no_worse_degree([H|T1],[H|T2]) :- no_worse_degree(T1,T2). /*
The predicate 'remove_subscripts(+Subs,+FreqIn,-FreqOut)' derives FreqOut by deleting any leading terms of FreqIn
that are embedded in Subs.
 */ remove_subscripts([],Freq,Freq) :- true. /*
 */ remove_subscripts([Sub|Subs],[Sub|FreqIn],FreqOut) :-
         remove_subscripts(Subs,FreqIn,FreqOut). /*
 */ remove_subscripts([_|Subs],FreqIn,FreqOut) :-
         remove_subscripts(Subs,FreqIn,FreqOut). /*
End of Optimiser. */
```

13.6 The Generator

The *Generator* may be invoked by a goal of the form:

```
        generate(Optimal,Text).
```

where *Optimal* is the file containing the output of the *Optimiser*, and *Text* is the file that should contain the annotated specification created by the *Generator*.

```
/* The Generator.
*/ generate(Input1,Input2,Output) :-
      read_tree(Input1,Tree),
      read_processes(Input2,Procs,Graph),
      generate_output(Output,Tree,Procs,Graph). /*
*/ read_tree(Input,Term) :- seeing(User),see(Input),read(Term),seen,see(User). /*
*/ read_processes(Input,Procs,Graph) :-
      see(Input),read(Procs),read(Graph),seen,see(user). /*
*/ generate_output(Output,Tree,Procs,Graph) :-
      'system$push$display'(message,left,'Generating labelled code...','','',''),
      telling(User),tell(Output),
      number_processes(_,Procs,Procs1),
      label_graph(Procs1,Graph,Labelled),
      incidence_matrix(Labelled),
      generate_system(Tree,Procs1),
      told,tell(User),
      'system$pop$display'(message). /*
*/ generate_system(system(System,Packages,Generics,States,Events),Sort) :-
      abolish(current_num,2),
      nl,write('package body '),write(System),write(' is'),nl,
      generate_states(States,Sort),
      generate_events(Events,Sort),
      write('end '),write(System),write(';'),nl,nl. /*
```

```
*/ generate_states([],_). /*
*/ generate_states([state(Lex,Codom)|States],Sort) :-
     tab(1),generate_decl(Lex,Codom,Sort,Proc),
     write(';                     -- '),write(Proc),nl,
     generate_states(States,Sort). /*
*/ generate_decl(lex([Id|[]],N),Codom,Sort,ProcId) :- !,
     Id=_/_/Name,write(Name),write(':'),write(Codom),
     look_up_process_id(lex([Id|[]],N),Sort,ProcId). /*
*/ generate_decl(lex([Id|Dom],N),Codom,Sort,ProcId) :-
     Id=_/_/Name,write(Name),write(':array ('),
     generate_doms(Dom),
     write(') of '),write(Codom),
     dummy_list(*,Dom,Stars),
     look_up_process_id(lex([Id|Stars],N),Sort,ProcId). /*
*/ generate_doms([Dom|[]]) :- write(Dom). /*
*/ generate_doms([Dom1,Dom2|Doms]) :-
     write(Dom1),write(','),generate_doms([Dom2|Doms]). /*
*/ generate_events([],_). /*
*/ generate_events([Event|Events],Sort) :-
     generate_event(Event,Sort),
     generate_events(Events,Sort). /*
*/ generate_event(event(Event,Params,Locals,Body),Sort) :-
     write(' procedure '),write(Event),
     generate_parameters(Params,Sort),
     generate_locals(2,Locals,Sort),
     write(' begin'),nl,
     generate_stmt_list(2,Body,Sort),
     write(' end '),write(Event),write(';'),nl. /*
*/ generate_parameters([],_) :- write(' is'),nl. /*
*/ generate_parameters(Params,Sort) :-
     write('('),generate_params(Params,Sort). /*
*/ generate_params([param(Lex,Codom)|[]],Sort) :-
     generate_decl(Lex,Codom,Sort,Proc),
     write(') is                 -- '),write(Proc),nl. /*
*/ generate_params([param(Lex,Codom)|Params],Sort) :-
     generate_decl(Lex,Codom,Sort,Proc),
     write(';                    -- '),write(Proc),nl,tab(15),
     generate_params(Params,Sort). /*
*/ generate_locals(_,[],_). /*
*/ generate_locals(N,[local(Lex,Codom)|Locals],Sort) :-
     tab(N),generate_decl(Lex,Codom,Sort,Proc),
     write(';                    -- '),write(Proc),nl,
     generate_locals(N,Locals,Sort). /*
*/ generate_stmt(N,[],_) :- tab(N),write('null;'),nl. /*
*/ generate_stmt(N,[H|T],Sort) :-
     tab(N),write('begin'),nl,
     N1 is N+1,generate_stmt_list(N1,[H|T],Sort),
     tab(N),write('end;'),nl. /*
*/ generate_stmt_list(N,[],_) :- tab(N),write('null;'),nl. /*
*/ generate_stmt_list(N,[H|T],Sort) :-
     generate_stmt(N,H,Sort),
     generate_stmt_tail(N,T,Sort). /*
*/ generate_stmt_tail(_,[],_). /*
*/ generate_stmt_tail(N,[H|T],Sort) :-
     generate_stmt(N,H,Sort),
     generate_stmt_tail(N,T,Sort). /*
*/ generate_stmt(N,if(lex([ExpName],1),Exp,True,False),Sort) :-
     look_up_process_id(lex([ExpName],1),Sort,ProcId),
     tab(N),write('if '),generate_expression(Exp),
     write(' then             -- '),write(ProcId),nl,
     N1 is N+1,
     generate_stmt_list(N1,True,Sort),
     generate_else_part(N,False,Sort),
     tab(N),write('end if;'),nl. /*
*/ generate_else_part(_,[],_) :- !. /*
*/ generate_else_part(N,False,Sort) :-
     tab(N),write('else'),nl,
     N1 is N+1,
     generate_stmt_list(N1,False,Sort). /*
```

```
*/ generate_stmt(N,while(Var,Exp,Body),Sort) :-
      look_up_process_id(Var,Sort,ProcId),
      tab(N),write('while '),generate_expression(Exp),
      write(' loop          -- '),write(ProcId),nl,
      N1 is N+1,generate_stmt_list(N1,Body,Sort),
      tab(N),write('end loop;'),nl. /*
*/ generate_stmt(N,all(Var,Dom,Body),Sort) :-
      look_up_process_id(Var,Sort,ProcId),
      Var=lex([_/_/Id],_),
      tab(N),write('all '),write(Id),write(' in '),write(Dom),
      write(' loop          -- '),write(ProcId),nl,
      N1 is N+1,generate_stmt_list(N1,Body,Sort),
      tab(N),write('end loop;'),nl. /*
*/ generate_stmt(N,for(Var,Dom,Body),Sort) :-
      look_up_process_id(Var,Sort,ProcId),
      Var=lex([_/_/Id],_),
      tab(N),write('for '),write(Id),write(' in '),write(Dom),
      write(' loop          -- '),write(ProcId),nl,
      N1 is N+1,generate_stmt_list(N1,Body,Sort),
      tab(N),write('end loop;'),nl. /*
*/ generate_stmt(N,assign(Var,Exp),Sort) :-
      look_up_process_id(Var,Sort,ProcId),
      tab(N),generate_var(Var),write(' := '),
      generate_expression(Exp),
      write(';   -- '),write(ProcId),nl. /*
*/ generate_stmt(N,call(Var,Exp),Sort) :-
      Var=lex([_/Package/Event],_),
      look_up_process_id(Var,Sort,ProcId),
      tab(N),write(Package),write('.'),write(Event),
      write('('),generate_expression(Exp),
      write(');   -- '),write(ProcId),nl. /*
*/ generate_stmt(N,null,_) :- tab(N),write('null;'),nl. /*
*/ generate_stmt(N,return([]),_) :- tab(N),write('return;'),nl. /*
*/ generate_stmt(N,declare(Vars,Body),Sort) :-
      tab(N),write('declare'),nl,
      N1 is N+1,
      generate_locals(N1,Vars,Sort),
      tab(N),write('begin'),nl,
      generate_stmt_list(N1,Body,Sort),
      tab(N),write('end;'),nl. /*
*/ generate_expression([]) :- !,generate_fn(Fn),write(Fn). /*
*/ generate_expression(Exp) :-
      generate_fn(Fn),write(Fn),write('('),
      generate_expn(Exp),write(')'). /*
*/ generate_expn([]) :- !. /*
*/ generate_expn([Var|[]]) :- !,
      generate_var(Var). /*
*/ generate_expn([Var|Exp]) :- !,
      generate_var(Var),write(','),
      generate_expn(Exp). /*
*/ generate_var(lex([_/_/Id|[]],_)) :- !,write(Id). /*
*/ generate_var(lex([_/_/Id|Subs],_)) :-
      write(Id),write('('),generate_subs(Subs),write(')'). /*
*/ generate_subs([_/_/Sub|[]]) :- write(Sub). /*
*/ generate_subs([_/_/Sub1,Sub2|Subs]) :-
      write(Sub1),write(','),generate_subs([Sub2|Subs]). /*
*/ label_graph(Procs,GraphIn,GraphOut) :-
      label_vertices(Procs,GraphIn,Graph1),
      sort(Graph1,GraphOut). /*
*/ label_vertices(_,[],[]) :- !. /*
*/ label_vertices(Procs,[Vertex-Edges|GraphIn],[Label-EdgeLabels|GraphOut]) :-
      label_vertex(Procs,Vertex,Label),
      label_edges(Procs,Edges,Labels),
      sort(Labels,EdgeLabels),
      label_vertices(Procs,GraphIn,GraphOut). /*
*/ label_vertex([(N,P,Vertex)|Procs],(P,Vertex),(N,P)) :- !. /*
*/ label_vertex([_|Procs],Vertex,Label) :-
      label_vertex(Procs,Vertex,Label). /*
*/ label_edges(_,[],[]) :- !. /*
*/ label_edges(Procs,[Edge|Edges],[Label|Labels]) :-
      label_vertex(Procs,Edge,Label),
      label_edges(Procs,Edges,Labels). /*
```

```
 */ look_up_process_id(Var,[(N,P,Vars)|Procs],(N,P)) :- member(Var,Vars),!. /*
 */ look_up_process_id(Var,[_|Procs],N) :- look_up_process_id(Var,Procs,N),!. /*
 */ look_up_process_id(Var,[],'(unassigned)'). /*
 */ generate_fn(Fn) :-
       unique(fn_,N),name(N,Digits),append("fn_",Digits,Name),name(Fn,Name). /*
 */ number_processes(0,[],[]). /*
 */ number_processes(N,[(Degree,Defs)|ProcsIn],[(ProcId,Degree,Defs)|ProcsOut]) :-
       number_processes(N1,ProcsIn,ProcsOut),N is N1+1,
       name(N,Suffix),append("Process_",Suffix,Id),name(ProcId,Id). /*
End of Generator. */
```