The Transformational Implementation of JSD Process Specifications via Finite Automata Representation

Andrew Paul Bass

Doctor of Philosophy

The University of Aston in Birmingham

September 1992

The University of Aston in Birmingham

The Transformational Implementation of JSD Process Specifications via Finite Automata Representation

Andrew Paul Bass

Doctor of Philosophy
September 1992

Summary

Conventional structured methods of software engineering are often based on the use of functional decomposition coupled with the Waterfall development process model. This approach is argued to be inadequate for coping with the evolutionary nature of large software systems. Alternative development paradigms, including the operational paradigm and the transformational paradigm, have been proposed to address the inadequacies of this conventional view of software development, and these are reviewed. JSD is presented as an example of an operational approach to software engineering, and is contrasted with other well documented examples. The thesis shows how aspects of JSD can be characterised with reference to formal language theory and automata theory. In particular, it is noted that Jackson structure diagrams are equivalent to regular expressions and can be thought of as specifying corresponding finite automata. The thesis discusses the automatic transformation of structure diagrams into finite automata using an algorithm adapted from compiler theory, and then extends the technique to deal with areas of JSD which are not strictly formalisable in terms of regular languages. In particular, an elegant and novel method for dealing with so called recognition (or parsing) difficulties is described. Various applications of the extended technique are described. They include a new method of automatically implementing the dismemberment transformation; an efficient way of implementing inversion in languages lacking a goto-statement; and a new in-the-large implementation strategy.

Keywords: Operational paradigm; transformational implementation; Jackson System Development (JSD); finite automata; computer-aided software engineering (CASE).

To my Family

Acknowledgements

I am greatly indebted to my supervisor, Bryan Ratcliff, not only for his guidance and insightful technical feedback, but also for his financial support during the period of this research, and in particular at a time of great uncertainty when I found myself between full-time contracts just as I was starting to write up. I feel that it is unlikely that I would have been able to complete the thesis without this support.

In addition, I would like to thank the following people for invaluable discussions and comments: fellow PhD students, Colin Lewis & Tony Rollo; Maeve Boyle, my colleague on the first phase of the PRESTIGE project, and John Aspinall and Graeme Walters who followed myself and Maeve on phase 2; John Cameron, Michael Jackson and Pam Zave who each provided insight and encouragement in the early stages.

Thanks are due to Dr. Rob Stammers of Aston Business School for the time he has allowed me to complete the write-up.

While gratefully acknowledging at those mentioned above, together with friends whose support kept me going while completion seemed to remain for so long over the horizon, I must of course take full responsibility for all errors and omissions which may remain in the thesis.

Contents

List of Figures

List of Tables

Chapter 1
Introduction and Overview

"I'd rather write programs to write programs than write programs"
Dick Sites
Digital Equipment Corporation†

1.1 Machines, Models and Methods

This thesis is motivated by three striking qualities of computer software:

1) its ability to describe (and manipulate) itself;
2) its ability to describe most other things (at least at some level of abstraction!);
3) its potential complexity.

The first, self-referential, quality of software may appear differently depending on the perspective of the observer. This quality is perhaps most explicit in the design of Turing's Universal Machine (Minsky, 1967), but it is central to many varied areas of computing, including programming languages (Ghezzi & Jazayeri, 1987) areas of artificial intelligence (Rich, 1990), and computer-aided software engineering (Hall, 1991). Guy Steele has said that "today's program is tomorrow's data"†. The thesis presents algorithms which take one representation of a software system and transform it into other representations.

Using software to describe things is often called *modelling*. Modelling is an idea which permeates computing; indeed, Minsky (*ibid.*) characterises digital computers themselves as "models for ...[the abstract mathematical concept of]... effective computability". Software is used to model economies, the weather, and the flight characteristics of airliners. In this work, attention is focused on its use in modelling the subject matter of information- and embedded- systems.

It is generally accepted that the development of large software systems depends on a methodical approach to the management of complexity (for example, Boehm, 1976; Dijkstra, 1976; Yourdon & Constantine, 1979; Jackson, 1983; Hoare, 1984; Booch, 1991) even if there is less agreement as to what form such a method should take. This

†(i) Quoted by John Bentley in "More Programming Pearls", Reading, Mass: Addison-Wesley, 1987
†(ii) In Bentley (*Ibid.*).

thesis draws on the methodology of the operational and transformational paradigms. Proponents of these paradigms argue that software should be specified with the intention of modelling the world in which it is to operate, and then should be automatically (i.e., under the direction of still more software) converted into a form in which it can be animated†.

1.2 Contributing Domains

This section briefly introduces those domains which underpin the thesis.

The Operational Paradigm

The functional hierarchy is a pervasive organising architecture for computer software systems. A major problem with such an architecture is that it is directed towards some abstract overall function which often has no analogue in the environment of the organisation (Cameron, 1986). Operational specifications directly represent the environment of an application and use its behaviour as the basis of the prescription of the intended behaviour of the application (Zave, 1984; Balzer, 1982). The key feature of an operational specification is its ability to generate the behaviour of the system it specifies. This characteristic affords various advantages, including the ability of such a specification to serve as a prototype, and the possibility of automatic implementation by correctness-preserving transformation.

There exist few methods in the operational paradigm. Perhaps the best documented examples are PAISLey (Zave, 1982; Zave, 1991), Gist (Balzer, 1982), Me-too (Henderson, 1986; Alexander & Jones, 1991) and Jackson System Development (JSD) (Jackson, 1983; Cameron, 1986). Of these, JSD has had the most commercial usage, and it is this method with which the present work is concerned.

Transformation Systems

Transformation systems (Partsch & Steinbrüggen, 1983) take well-formed behaviour-producing representations and manipulate them to obtain different representations which generate the same behaviours. Typically, the antecedent and consequent representations will differ with respect to criteria such as clarity, speed of execution, and space complexity. Often qualities such as clarity and efficiency seem to be mutually exclusive — hence Jackson's first law of software optimisation "Don't do it"! (1975). By using a

† This description might be said to include the compilation of programs written in a high-level programming language. The distinction is one of abstraction level: the vocabulary of a computer program is general-purpose and consists of variables and operations, while the vocabulary of an operational model is application-specific and consists of features of the problem domain.

transformation system, it is possible to specify a system using a clear representation and then confidently convert it to an efficient one for subsequent execution.

Most research into transformation systems has concentrated on the manipulation of functional programs, as the strong algebraic properties of languages based on recursion equations make them highly amenable to such manipulation (Henderson, 1986). There has also been interest in transformations between programming paradigms, in particular, from the functional to the imperative paradigm (Partsch, 1991). This thesis is concerned mainly with a third class of transformation which can be said to be architectural in nature. Transformations of this sort take abstract operational specifications and produce concrete implementations (Jackson, 1983).

Automata and Language Theory

Automata theory and formal language theory are modern but well established branches of mathematics. Their development has been closely associated with the development of digital computers and computer programming (Minsky, 1967). Automata are, in fact, abstract computing devices whose interactions with their environments can be expressed using formal languages. These twin theoretical domains have many applications in managing the complexity of computer software by providing tractable abstract models of complex behaviour, and have been used in areas as diverse as programming language design and implementation, communication network synchronisation and pattern matching (Aho, *et al.*, 1985). They also underlie the techniques used in JSD for constructing models of behaviour in the environment of proposed software systems. The nature of the relationship between the formal theories and JSD is examined in this thesis, and is used to facilitate the automation of various transformations on JSD specifications.

1.3 Scope of Research

Aims and Objectives

The transformations usually used in the implementation of JSD specifications are for the most part presented in an informal and pragmatic way (for example, see Jackson, 1983; Cameron, 1986; Cameron, 1988). Furthermore, there have been few radical changes to the repertoire since Jackson's introductory book (*ibid.*). It seems likely that progress in discovering new transformations or new ways of automating existing transformations will come only through deeper examination of the theoretical ideas underlying JSD. Support for this notion comes from Sridhar & Hoare (1985) who show how the algebraic laws governing CSP can be used to transform simple JSD specifications. The connection between CSP and JSD has recently been further investigated by Yeung *et al.* (1991). The present work draws on Hughes' (1979) explication of the programming component of the Jackson methods, JSP, in terms of formal language theory. Language theory cannot provide a complete account of a notation for concurrent systems such as JSD (Milner, 1980), but because of the availability of a large body of knowledge concerning the processing of languages and automata, it nonetheless provides a point of leverage in the pragmatic development of transformations.

In summary, the main aim of this research has been the application of language theory to the design of automatable transformations for the efficient implementation of JSD specifications. To this end, the following objectives have been pursued:

- to make explicit the links between JSD and the interrelated theories of regular languages and finite automata;
- to design and implement a transformational approach to bridge the gap between the JSD world and the formal theories;
- to exploit the properties of the formal artifacts produced by the transformational approach to generate efficient implementations of JSD processes, in particular so-called *dismembered* implementations.

Related Work

This work has benefited from recent research by Lewis (1991) into the realisation of JSD specifications in object-oriented languages. Lewis used a language-theoretic concept, that of *follow sets*, as a novel way to implement JSD specifications in a Smalltalk-80 environment. This thesis takes the link with language theory further, and leads to the explicit construction of finite automata from JSD specification elements. In doing so, it is able to generate significantly more efficient and elegant implementations than Lewis's approach, as well as to address special cases he did not cover.

One piece of research which is closely related to this work is an approach to the automatic dismemberment of process structures proposed by Cameron (1990). Unlike the present approach, Cameron does not construct any intermediate representations between process structures and program texts. Rather he defines permissible dismemberments in terms of an algebra of process structure trees. The approach is very elegant and seems likely to yield implementations of equivalent efficiency to those discussed here. However, the results of this work have not been published or otherwise made available† . Furthermore, Cameron's ideas are specifically aimed at automating the dismemberment

† As of Summer 1992

transformation. While the present work shares this aim, it has produced an approach which may have wider applications (discussed in Chapter 8).

Areas Not Covered

Milner (1980) discusses the limitations in using a finite automata model of communicating systems, and these limitations apply in the case of JSD (Yeung *et al.*, 1991). No attempt has therefore been made to model the total semantics of JSD with a communicating automata model. This work is more concerned with individual processes than with their communication, which is dealt with pragmatically in a characteristically JSD style.

As the focus of the work is on an event-based view of JSD, less attention has been paid to considerations of state-vector (database) management. So, for example, although the need to enforce mutual exclusive access to state-vectors is mentioned in connection with dismemberment (Chapter 6), detailed discussion of the means to achieving this requirement is not given.

The ability to automate dismemberment transformations opens up the possibility of applying a new implementation strategy to JSD networks. This strategy, which is called *network dismemberment*, is introduced in Chapter 6. As is discussed in Chapter 8, much work remains to be done to understand fully the utilisation of this implementation strategy, and it is given only restricted treatment here.

Structure of the Thesis

The thesis is divided into eight chapters. Chapter 2 discusses two important software engineering paradigms which underpin JSD: the operational paradigm and the transformational paradigm. The introduction of these ideas is motivated by a discussion of software evolution and the inadequacies of conventional structured methods in coping with fluid system requirements. JSD is introduced as an example of an operational approach to software engineering, and is contrasted with other well documented examples.

Chapter 3 presents elements of formal language theory and its relationship to automata theory using the notations of JSD. In particular, Jackson trees are related to regular expressions and their corresponding finite automata. Examples from typical JSD application domains are presented. In adopting this approach, all three fields are introduced together. This chapter establishes the foundations upon which the research is based.

Chapter 4 discusses the automatic transformation of Jackson trees into corresponding finite automata, using an algorithm adapted from ideas developed in compiler theory. This begins the systematic development of the transformational approach contributed by the thesis.

Real JSD specifications generally feature regular expressions *augmented* with other elements. Chapter 5 extends the approach developed in the preceding chapter to deal with these augmentations, thus generalising the applicability of the approach. In particular, an elegant and novel method for dealing with so called *recognition* (or *parsing*) *difficulties* is described.

Chapter 6 shows how the approach developed can be used to support the generation of efficient implementations of JSD specifications in ways previously only achievable by hand. This is argued to extend significantly the applicability of such implementation techniques.

Chapter 7 discusses the practical work associated with the thesis. It describes the implementation of the algorithms developed in this work. These algorithms have been incorporated into an experimental version of an integrated JSD implementor's toolkit developed by the author and others.

The concluding chapter summarises the main points of the thesis and critically evaluates what has been achieved. Finally, suggestions are offered for further work to extend the usefulness and applicability of the results obtained.

Chapter 2

The Operational Paradigm in Software Engineering

This chapter establishes the context in which the work of the thesis appears. It identifies flaws in conventional software development methodology, and argues that an alternative approach, based on the construction and automatic manipulation of executable specifications, substantially addresses these weaknesses.

2.1 Software Evolution and System Development Methods

Among the many difficulties in constructing high-quality solutions to real-world software problems are the following:

The requirements for a system are difficult to capture
- users may be only partially aware of their requirements
- communication between users and developers may be made difficult by differences in domain and computing jargon.

The requirements are volatile and may be revised to include
- clarifications of incorrectly captured requirements
- new requirements suggested by experience of exercising the implementation.

These factors tend to contribute to a perceived mismatch between the user's expectations and the actual behaviour of a system, if not immediately, then as the system is used in its intended environment. As Jackson & McCraken (1981) note: "any system development activity inevitably changes the environment out of which the need for the system arose". As a change in the environment of a system is likely to invalidate portions of its functionality, it can be concluded that the very activity of developing a software system will give rise to the need for modifications to the original specification (Lehman, 1991). Unfortunately, as software is modified it typically becomes less structured, more error-prone and less amenable to subsequent change, until eventually a system may be so atrophied that it must be discarded and replaced by an entirely new system (Lehman & Belady, 1980). Giddings (1984) has coined the term 'Domain Dependent' to describe software implementing large systems such as business systems, operating systems and embedded systems:

> "[Domain Dependent] software is characterised by interdependence between the universe of discourse and the software. The use of the software may change both the form and the substance of the universe of discourse, and as a result, the nature of the problem being solved".

The dynamics of software changes described above are exacerbated by the perceived *changeability* of computer software (Brooks, 1987) which encourages both users and developers to suggest and attempt (perhaps unnecessary) revisions. Brooks describes software as being "pure thought-stuff, infinitely malleable", and argues that this malleability leads to the perception that it is easy to change. The lack of *physical* resources needed to change software adds to the illusion (Lehman, 1991).

Clearly, a large part of software engineering effort will be concerned with the ongoing revision of software to keep up with the changing requirements of the user. The activity of changing software to keep up with drifting requirements is usually called 'maintenance' although Lehman & Belady (1980) considers the terms misleading, prefering instead 'program evolution'. The difference in the terms demonstrates the difference in perspective between researchers and many practitioners. Schneidewind (1987) takes the view that change should be regarded as an integral part of a development process, while the reality is that it is often appended as an afterthought. Whatever its name, there is no doubt concerning its economic significance. Lientz and Swanson (1980) found in a major survey that many North American organisations were spending between 20% and 70% of their software resource on maintenance activity.

It is clear that a successful approach to system development should comfortably accommodate an almost inevitable drift in functional requirements. The conventional approach, by which is here meant a combination of *top down functional decomposition* (TDFD) and the *Waterfall* software process model, has frequently been criticised for failing in this respect. The following discussion summarise the chief objections.

Critique of the Waterfall Software Development Process Model

The Waterfall software development process model (e.g. Royce, 1970; Boehm, 1976) is generally accepted as the conventional process model for software development (Lehman, 1991). The same basic model has appeared in various guises over the years, but Boehm's formulation shown in Figure 2.1, is representative.

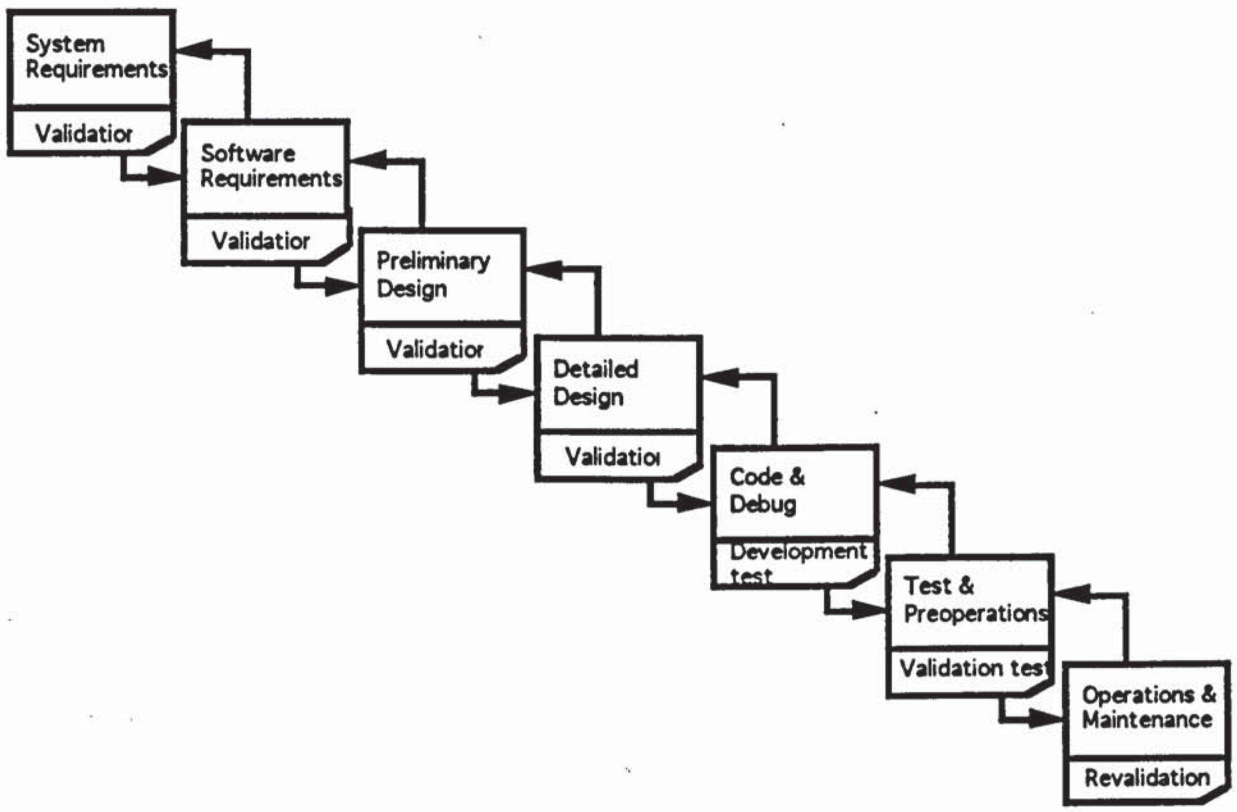


Figure 2.1. The Waterfall model of software development, after Boehm (1976).

The Waterfall approach attempts to manage the complexity of a development process by separating specification (often defined as 'the WHAT') from implementation (the 'HOW')†. It is only after the specification and design have been validated and signed off that the move is made to implementation. All decisions regarding the binding of functions to resources are committed early (Zave, 1984). This practice developed when access to computers was scarce; the high level of structure in the early stages was intended to reduce time wasted by unnecessary debugging in the code and test phases. Even now, with computing resources much more freely available, it has been argued that it still suits project managers to follow a Waterfall model for reasons that include the following:

- the requirements specification can form the basis for a development contract (Hall, 1991b);
- accountability is high during the process as well-defined milestones and corresponding intermediate deliverables can be agreed *a priori* (Sommerville, 1991);
- the idea of separating specification from implementation, perhaps as a result of

†Although the *Specification/WHAT, Implementation/HOW* dichotomy is considerably overworked, its use seems unavoidable here, as the subsequent discussion introduces an alternative view of software development based on an explicit change of these bindings to *Specification/PROBLEM-ORIENTED, Implementation/SOLUTION-ORIENTED*.

intuitive appeal, has become the conventional wisdom (Zave, 1984).

The first two reasons have more commercial, than methodological, utility, since contracts and milestones can be used to resolve the almost inevitable disputes which arise as requirements change, deadlines are missed and budgets are overrun. They characterise a fundamentally negative, if natural, response to the problems of software evolution in which rather than aim to produce good software, those concerned work to ensure that they will not be blamed when things go wrong.

The third reason offered for the continuing popularity of the Waterfall model can also be criticised. Swartout and Balzer (1982) reject the idea of a strict separation between specification and implementation and argue that they must instead be accepted as being "inevitably intertwined". They view the software development process as the discovery of a hierarchy of specifications, considering that "every specification is an implementation of some other higher level specification". Swartout and Balzer argue that in the process of implementation, there will always tend to be the need to revise retrospectively superordinate specifications in the hierarchy. Specifically, they identify the following classes of revision:

- to account for the realities and limitations of the implementation environment (generally in regard to resource availability);
- to accommodate new insights about requirements, interactions between software components, etc.

McCraken and Jackson (1981), Boehm (1988) and Schneidewind (1987) are among writers who identify other problems with the Waterfall model, including:

- late extant system behaviour (meaning late feedback regarding incorrectly captured requirement);
- failure to accommodate the use of potentially time and cost saving techniques such as prototyping and transformations;
- difficulty of accommodating end-user development;
- lack of support for software evolution.

Critique of Top Down Functional Decomposition

By adopting the TDFD approach, a developer chooses to regard a system as implementing a complex function from a set of inputs to a set of outputs. Progressive refinement of the function leads to a program or system structure of hierarchically arranged subfunctions contributing to the calculation of the original functional requirement. Among its advantages are the following:

- complexity is managed through the production of an abstraction hierarchy (Dijkstra, 1976);
- at any stage in the refinement process, the design always encompasses a complete solution to the problem (Ratcliff, 1987)

Although it is an extremely popular approach, underlying many methods (for example, Yourdon & Constantine, 1979; DeMarco, 1978; Gomaa, 1984; Nicholls, 1987) TDFD has attracted much criticism. Among the major disadvantages identified by its detractors are the following:

- TDFD produces a system architecture based on an endemically volatile aspect of a user's problem, namely its functional requirements (Jackson, 1983);
- it can be very hard to make appropriate decompositions (Cameron, 1986);
- TDFD forces the developer to make the highest risk decisions (the initial high level decompositions) in the presence of the minimum pertinent information (Agresti, 1986).

These criticisms are expanded on below.

As has been seen earlier, many writers have argued that provision for system evolution should be regarded as an integral part of a software process model (e.g., Lehman & Belady, 1985). An important factor in making a system amenable to change is the architectural structure of the software (Parnas, 1971; Harrison & Magel, 1981). TDFD leads to system structures based on the decomposition of an overall functional requirement. This means that should the requirement be revised (which is almost inevitable) the entire abstraction hierarchy could potentially be invalidated. Jackson (1983) gives an example of a program developed using functional decomposition which is unable to accommodate a seemingly small drift in functional requirements without a major reimplementation.

The addition of data modelling techniques to TDFD, as for example in SSADM (Nicholls, 1987), has gone some way towards anchoring system designs to stable features of the problem domain. Here, data structures (and in the case of object-oriented databases, accessing methods) are based on enduring properties of the real world, namely the attributes of participating entities. Indeed, for database applications where processing consists purely of simple updates and queries, and flexibility can be provided by an *ad hoc* query language interpreter, this approach is successful. Avison (1987) is among those who have put forward system development methods based on this idea (although he more recently (1988) advocates this only a component of a larger 'contingency' framework). Data modelling is a helpful addition to TDFD, but still runs into problems where complex processing is necessary, as the *processing* structure of the system is still based on a decomposition of the original functional requirement.

Parnas's landmark paper "On the criteria for the decomposition of modules" (1971) was an early warning of the potential dangers of modularisation based on functional decomposition. Parnas demonstrates how alternative decompositions of the same specification lead to systems with very different modifiability, and offers some criteria for identifying good decompositions. In a paper published fifteen years later, Cameron (1986) criticises TDFD for precisely the reason that it is very difficult to make the correct choices first time. In fact, the highest risk decisions — those regarding overall system structure — are made very early in the process, and the insight necessary to make good decompositions only comes as the implementation level is approached. The risk-orientated viewpoint is taken further by Boehm as the motivation behind his Spiral model of software development (1988). The Spiral model is a software process meta-model which is instantiated according to the risk structure of a particular problem. Using this model, Boehm identifies the TDFD/Waterfall approach as a suitable development process when the problem is well-understood and requirements are stable (as would be the case for a straightforward payroll system, for example). In such a situation, where there is low risk of acquiring fundamental misconceptions about the problem or of receiving late change requests from user, the developer benefits from the predictability of an orderly Waterfall life-cycle. Conversely, in the high-risk context created by the dynamics of large, novel endeavours, such a process model is highly inappropriate.

Although he is very critical of TDFD as a design method, Jackson (1983) acknowledges hierarchical decomposition to be a good description technique. This may account for the enduring success of top-down methodology — textbook examples of TDFD are very neat and easy to follow, and the reader is encouraged to believe that rigorous application of the method will work similarly for new problems. In truth, the examples have almost certainly been constructed retrospectively. Practitioners, keen to demonstrate that they are adhering to good software engineering practice, may tend to perpetuate the fallacy that TDFD is a good way to design systems, as hinted by Parnas & Clements (1986) in their paper "A rational design process: how and why to fake it"(!). (Actually, Parnas & Clements recognise some real value in describing a finished system in terms of how it should have been built, arguing that at least it provides a basis for maintenance.)

TDFD has been adapted to the particular problems of implementing systems featuring concurrent processing (Gomaa, 1984; Neilson & Shumate, 1987). However, separate threads of control are not explicitly considered until after the functional requirements have been decomposed into a system of dataflow through processing elements. Only then are these individual elements composed into 'tasks'. These tasks are unlikely to reflect the real-world concurrency which drives the system (Sanden, 1989). As a result, two or more real-world threads of activity may require shared access to code exported by a single task, which then has to handle the concurrency internally.

The Relationship between the Waterfall Model and TDFD

TDFD requires the developer to take high risk decisions (regarding, for example, functional requirements and appropriate decompositions) early in the development process (Boehm, 1988). As testable behaviour-producing objects appear only at the lowest level of the decomposition, errors in early decisions are often not discovered until much money and time has been spent on design and coding. Backtracking at this stage may be prohibitively expensive. Project managers therefore attempt to ensure that development decisions are made correctly first time, by rigorously applying a Waterfall model, complete with attendant paperwork to be signed off as work proceeds. Boehm (1988) and Agresti (1986) are among those who point out that this is an effective strategy only when the functional requirement is stable and the problem is well understood. In other cases, as has been observed, there are many problems with such a rigid approach.

2.2 The Operational Paradigm

The operational paradigm (Zave, 1984; Agresti, 1986) has been put forward as an alternative to the conventional approach discussed above. An operational specification takes the form of a *model* of the proposed application *in its environment* — that is, portions of the real-world problem domain are explicitly represented as fundamental components of the specification — and can generate the behaviour of the system it specifies, either by direct execution, or after some form of translation (Balzer & Goldman, 1986). An operational model is made up of implementation independent structures, which have the potential to be realised in a variety of ways. Choice of factors such as programming language, operating environment (hardware and software) and database organisation remain essentially unconstrained. There can also be any number of threads of control. During specification, the developer is unconcerned with efficiency; production of the required behaviour is the sole aim. As a result, operational specifications tend to execute too slowly to serve as anything other than early prototypes (Zave, 1984). The implementation of an operational specification involves the definition of constraints on the behaviour-producing mechanisms which can be used to meet the specification, and on the space and time resources which are available, followed by the (automated) *transformation* of the specification to meet these constraints. Note that a transformational approach to implementation is possible because of the executable semantics of operational specifications. Ideally, software tool support should exist to perform the transformations (see Section 2.3).

Reframing the Separation of Concerns

The idea of *separation of concerns* is invaluable in managing the complexity of software development. It is the principle behind Parnas's suggestions regarding information hiding (1971), stepwise refinement (Wirth, 1971), object-oriented programming and design (Cox, 1986; Booch, 1991) etc. However, one must be careful in selecting which factors are separated. Critics of conventional software engineering methodology argue that the WHAT—HOW axis is not a feasible choice, (Swartout & Balzer, 1982) but rather that the PROBLEM-ORIENTED—IMPLEMENTATION-ORIENTED axis is more appropriate. For example,

> "The structure of the operational specification is problem-oriented but not implementation-oriented. During the transformation phase the specification is subjected to transformations that preserve external behaviour, but alter or augment the mechanisms by which that behaviour is produced, so as to yield an implementation-oriented specification of the same system" (Zave, 1984) .

This reframing is a key insight of the operational approach. It is summarised in Figure 2.2.

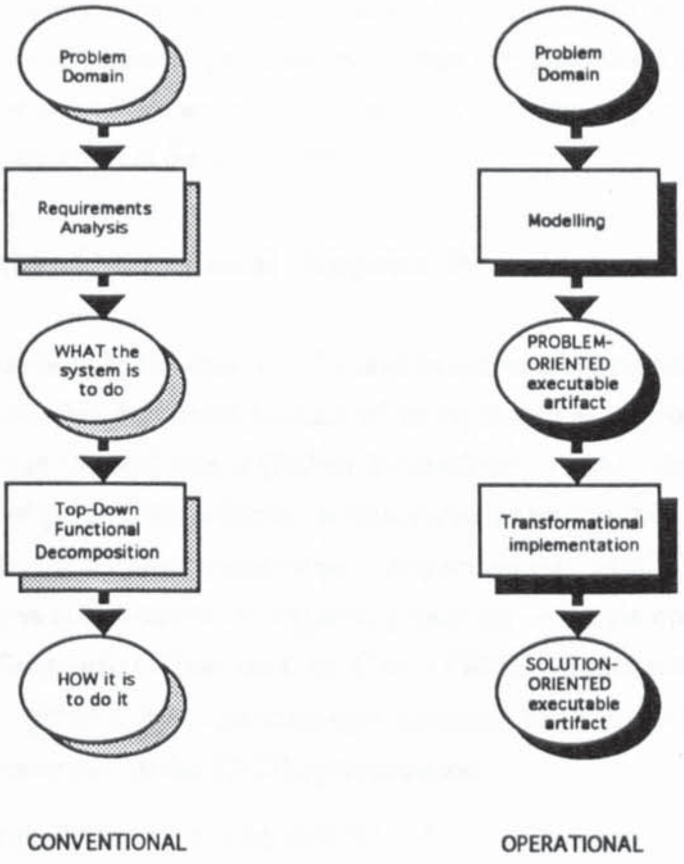

Problem Domain
Requirements Analysis
WHAT the system is to do
Top-Down Functional Decomposition
HOW it is to do it
CONVENTIONAL
Problem Domain
Modelling
PROBLEM-ORIENTED executable artifact
Transformational implementation
SOLUTION-ORIENTED executable artifact
OPERATIONAL

Figure 2.2. Separation of concerns in conventional and operational paradigms.

There is general agreement that a specification should not lead a developer towards a particular implementation (see for example Ratcliff, 1987). Conventional methodology takes this to mean that a specification should describe the behaviour of a black box, accepting stimuli and producing responses, and should say nothing about the box's internal structure. This is predicated on the belief that an explicit structure must bias the form of an implementation (Zave, 1982). The operational paradigm is liberated from this assumption by the behaviour-preserving restructuring capabilities offered by transformation. Operational specifications are therefore able to incorporate descriptions of the structure of behaviour-producing mechanisms in order to express executable semantics. Executability is a key feature of operational specifications as it offers the possibility of

- prototyping (Zave & Schell, 1986; Balzer *et al.*, 1982; Warhurst *et al.*, 1990);
- transformational implementation (Agresti, 1986);
- increased insight into the problem domain through modelling of its dynamic behaviour (Renold, 1988b).

While the internal structure of an operational specification is explicit, rather than hidden in a yet-to-be-decomposed black box, it is argued not to constitute the *design* of a particular system, since it makes no reference to any particular runtime environment (Zave, 1984). Indeed, one of the great appeals of the approach is that an operational specification can be implemented in a variety of environments simply by applying different transformations. Automata theory (see Chapter 3) provides more than adequate support to the proposition that a system's internal structure can be described without suggesting any particular implementation (Rockstrom & Saracco, 1982).

Examples of the Operational Approach

Below are described four of the best documented operational approaches: JSD, PAISLey, GIST, and Me-Too. An important feature of an operational approach is its ability to capture parallelism in the real world (Balzer & Goldman, 1986). Zave & Schell (1986) identify two forms of parallelism inherent in real-world software problems which they term *asynchronous* and *synchronous* parallelism. Asynchronous parallelism is specified by putting asynchronous computations in separate processes— a style epitomised by Hoare's Communicating Sequential Processes or CSP (1985). Synchronous parallelism is expressed within a process by exploiting opportunities to evaluate intermediate values concurrently. For example, in the PAISLey expression

```
sq-rt[r-sum[(square[a],square[b])]]
```

(which has the value of the length of the hypotenuse of a right-angled triangle with sides of length a and b), the subexpressions `square[a]` and `square[b]` can be evaluated in parallel. This kind of behaviour is termed as synchronous because the evaluations are synchronised at their endpoints.

The four methods discussed below vary in their ability to capture parallelism of either form. These differences are summarised in Table 2.1.

	Asynchronous	**Synchronous**
JSD	Yes	Not explicitly
PAISLey	Yes	Yes
GIST	Yes	Yes
Me-Too	No	Yes, by implication

Table 2.1. Comparison of operational methods with respect to types of parallelism expressible.

JSD and PAISLey are more closely related to each other than to the other methods, particularly through their explicit use of sequential processes to represent asynchronous behaviours. Such processes have the following advantages:

- the ability to capture naturally asynchronous processes in the real world (Jackson, 1983) — this is especially important in real-time applications and is becoming more so in data processing with the proliferation of distributed and on-line transaction systems;
- the ability to encapsulate state, and therefore allocate responsibility for the maintenance of its integrity to a well-defined part of the system;
- the partitioning of system behaviour into sequential programs which are individually easy to understand (Sanden, 1989).

Use of the process abstraction is not, of course, limited to those methods which are generally characterised as operational. Among the notations which employ sequential processes as a specification element are CSP (Hoare, 1985), CCS (Milner, 1980), and SDL (Rockstrom & Saracco, 1982).

JSD

A little more space is devoted to JSD than the other approaches, as it forms the subject matter of the rest of the thesis. The description here is nevertheless brief and informal. A deeper characterisation in terms of language and automata theory is given in the next chapter.

JSD bases the structure of a system around a model of stable aspects of the real world (Jackson, 1983), namely *entities* in the problem domain and the *actions* in which they participate — an approach that can be described as *entity life history modelling* (Sanden, 1989). This style of modelling is distinct from static data modelling due to a strong emphasis on the time ordering of actions. Legal orderings of actions are imposed by associating with each entity a structure, expressed in a graphical notation equivalent to regular expressions (Hughes, 1979), which reflects the life history of the corresponding real world object. For example, Figure 2.3 could describe the structure of a bank ACCOUNT.

The constraint provided by these time ordering specifications ensures that updates to the state of the equivalent model in the system — a set of 'long-running' sequential processes — can occur only in a fashion consistent with changes in the real world. These updates are initiated by inputs to the system resulting from occurrences of actions in the real world. The system model thus tracks the progress of entities as they perform or experience these actions. Providing an inevitable time delay is accepted between a change in the real world and corresponding system update, the model processes will reflect the state of the real world at any point in time (at some abstract level).

Once the entities in a system have been described as model processes, *function processes* are added to meet functional requirements. There are two basic types of function:

- those which answer questions about the current state of the real world;
- those which answer questions about the history of the real world.

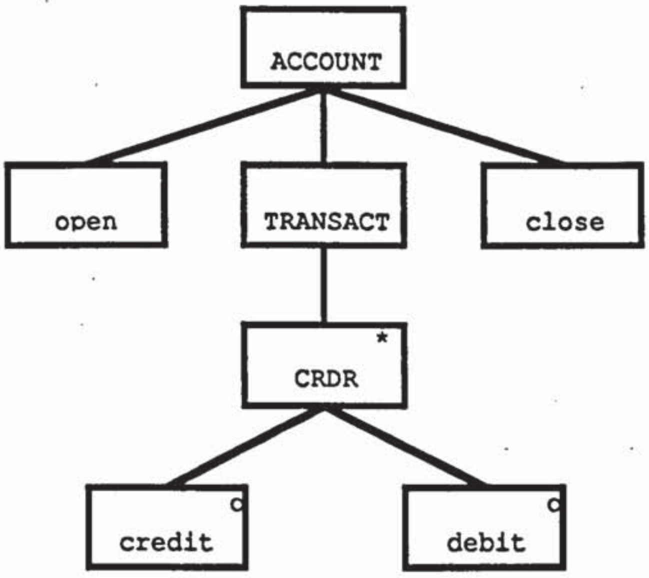


Figure 2.3. Structure of an ACCOUNT within a bank system.

To facilitate the answering of these questions, function processes must be connected to the model using one of two interprocess connections: *state-vector inspections* and *datastream connections* (more complex connections exist but can be ignored for current purposes).

The persistent state of a model process is known as its *state vector*. Assuming three domains for illustrative purposes — banking, warehousing and air-traffic control — state-vector inspections (read-only accesses — see Figure 2.4) will then support functions such as:

- List all overdrawn accounts.
- How many Snickers Bars are in the stores?
- Are any aircraft on collision courses?

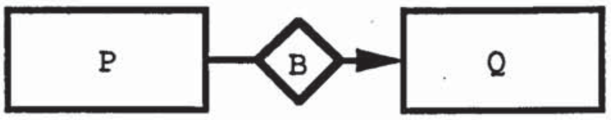


Figure 2.4. An example of state-vector connection (P inspects Q).

As the model changes, information can be generated about what has happened. Each time an event of interest occurs, a record can be inserted in a datastream which is read by a function process (see Figure 2.5).

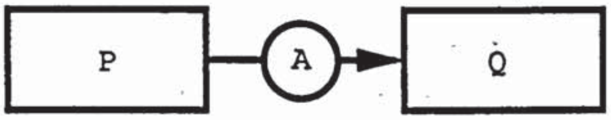


Figure 2.5. An example of datastream connection (P writes to Q).

Datastreams provide the information to support functions such as:

- List all withdrawals over £1000 since the last request.
- What has been the average daily consumption of Snickers bars since their name was changed?
- How many 'near misses' have occurred in UK air space in the last year?

Connecting function processes to the model produces a network of asynchronous communicating sequential processes. This network, together with the internal structures of the processes, comprises a (probably inefficient) executable specification of the system. To produce an efficient implementation, the JSD implementor applies transformations from a standard repertoire. We describe these in the next chapter, and go on in the rest of the thesis to develop the implementation of a particular class of transformations, *dismemberments*, for inclusion in the PRESTIGE JSD transformational workbench (Bass *et al.*, 1991; Ratcliff, 1990).

PAISLey

Zave (1982) introduces the PAISLey language for the operational specification of embedded computer systems. PAISLey is an acronym for Process-oriented Applicative Interpretable Specification Language. As with JSD, the key unit of specification in PAISLey is the asynchronous process. In considering the internal behaviour of processes, PAISLey like JSD, models entity life-histories. Unlike JSD, however, the style which Zave advocates for expressing a process's activity is applicative. Among the advantages she puts forward for such a style are their interpretability, manipulability and "tremendous powers of abstraction", which allow computations to be specified without constraining the order of evaluation of intermediate results, the allocation of storage or the underlying representation of data. By combining functional programming with asynchronous processes, PAISLey specifications can exhibit maximal parallelism for the problem being described (Zave & Schell, 1986).

Processes and process states, are not, of course, applicative concepts. They are judged necessary in PAISLey because such aspects as performance requirements and real-time interfaces with the environment are difficult to specify applicatively. Processes are specified by supplying a state space and a *successor function* on that state-space (i.e., a function which when applied to one state in the state space will deliver another). A process goes through an infinite cycle of states asynchronously with respect to all other processes. The point at which the applicative and nonapplicative elements of PAISLey meet is in the mechanism for interprocess communication. Processes communicate by applying *exchange functions* which seem locally to be applicative, but have the global side-effects associated with exchange of data. A choice of exchange function types is available to the specifier, offering rendezvous-like semantics with a variety of priority and timeout characteristics.

PAISLey enables the specifier to associate *timing attributes* with functions. These specify evaluation-time ranges and distributions which can be used when a specification is executed to simulate its performance characteristics. *Reliability attributes*, in the form of probabilities of evaluation failure, can also be assigned to functions to enable the simulation of unreliable components and interfaces.

Zave (1991a) summarises the PAISLey project and extracts general lessons learned over the project lifetime. The PAISLey system is available from AT&T Bell Laboratories for educational use (Zave, 1991b). It is implemented in the C programming language and supports the editing and execution of PAISLey specifications.

GIST

The GIST language (Balzer, Goldman & Wile, 1982) is an operational specification language whose semantics are defined in terms of how state is modified by the operations comprising a specification. In this respect it has similarities with JSD and PAISLey. A GIST specification models a *closed system* incorporating agents considered to be external to the computerised subsystem. Therefore GIST specifications model *relevant aspects* of the whole system. Again, this is true for JSD and PAISLey.

GIST differs in flavour from JSD and PAISLey in the way behaviour is specified. The fundamental behaviour-producing element in a GIST specification is called a *demon*, and is essentially a stimulus-response rule which fires in response to particular patterns in a global database, causing one or more operations to be performed. The firing of a demon will generally lead to updates to the database which then may cause other demons to be enabled. When a demon's stimulus is enabled, its response becomes an active *line of control*. The non-deterministic interleaving of the activity specified by all active lines of control generates a space of possible behaviours which can ensue from a particular situation. In general, however, such a space will allow behaviours which are *not* required of the final system. The GIST specifier can place constraints on the permissible states of the system, and on the applicability of individual demons, and so 'prune' the specification until it generates only the behaviours required.

GIST specifications are, in principle, executable by a suitable production-rule interpreter. As the resulting performance would be intolerably slow, transformations are necessary to produce an acceptable implementation (Feather and Cohen, 1982). The large amount of nondeterminism in a GIST specification provides a corresponding amount of flexibility in scheduling the execution of operations.

Me Too

Me too (Henderson, 1986; Alexander & Jones, 1991) is an operational method which uses a language formed by a combination of the formal specification notation VDM (Jones, 1990) and a functional programming language related to Miranda (Turner, 1986). There are three steps to the method: Model, Specify, and Validate. Initially, a model is produced which comprises abstract data objects, corresponding to real-world entities, and associated operations. This model is then refined into a specification in the Me too language — objects become abstract data types, and their operations become functional definitions in terms of recursion equations. Me too is executable (by an interpreter available in several dialects of Lisp), so the specification can now be exercised to see if it yields the required behaviour. The results of this evaluation are fed back to the Model stage in an iterative manner. Alexander & Jones (1991) provide several examples of the application of Me too to software specification problems.

Transformation of Me too specifications into efficient implementations is not mentioned as a feature of the method, however, should an imperative program be required, then transformational techniques of the sort discussed later in the chapter would seem applicable.

Me too is an attractive method in that it combines the twin benefits of formality, allowing specifications to be reasoned about and manipulated, and executability, allowing user-validation of the appropriateness of the specification. Its main deficiency, which Alexander & Jones acknowledge, is the lack of support for the expression of asynchronous parallelism. This limits Me too's applicability for the modelling of many real-time and user-interface applications. However, ongoing experimentation with a CSP/Me too hybrid language may offer a promising way forward.

Summary

Table 2.2 summarises the differences between the conventional and operational approaches.

Conventional approach	Operational Approach
• Separates WHAT from HOW	• Separates PROBLEM-WORLD from IMPLEMENTATION-WORLD
• Intertwines PROBLEM and IMPLEMENTATION	• Intertwines WHAT and HOW
• Bases structure on functional decomposition of volatile requirements	• Bases structure on elaboration of stable real world model
• Behaviour seen late so need for strict control (Waterfall)	• Behaviour seen early so feedback early
• Concurrency hidden in modules	• Naturally models concurrency, so modules are sequential programs
• Implementation by design, code and test	• Implementation by automatic correctness preserving transformation

Table 2.2. Key differences between conventional and operational paradigms of software development.

This section has reviewed four of the best documented operational methods: JSD, PAISLey, GIST and Me too. These four methods are very different in flavour — the operational approach is not characterised by the kinds of notations employed. Among the formalisms which have been employed for expressing operational specifications are:

- regular structures, e.g. JSD, CSP (Sridhar & Hoare, 1985);
- production rules, e.g. GIST;
- recursion equations e.g. Me too;
- horn clauses e.g. (Davis, 1982);
- algebraic notations, e.g. OBJ (Goguen & Meseguer, 1982).

The common threads linking these seemingly diverse approaches are (1) modelling of the real-world, (2) executability, and (3) the potential for transformational implementation.

2.3 Transformational Implementation

Operational methods are dependant on the ability to transform specifications into efficient forms. This section looks at the area of program and design transformation. Program transformation systems (Partsch & Steinbrüggen, 1983; Partsch, 1991) take an executable specification which exhibits a desired behaviour, and change the mechanism for producing that behaviour. Transformations alter the trade-off between the clarity and efficiency of a program, while preserving its correctness. Compilation is an example of correctness-preserving transformation process, and like compilation, many transformations are amenable to automation.

The discussion below falls into three sections. In the first, some common transformational rules and techniques are introduced. These can be used for program improvement purposes or combined into complete implementation strategies. The second section considers the transformation of functional programs. Target implementations may be produced either in the same language as the original program, or in an imperative language such as Fortran. The third section considers the architectural transformations which are used in the implementation of JSD (similar techniques would be needed for the implementation of specifications expressed in any operational process-oriented form).

Types of Transformation Rule

Following Partsch & Steinbrüggen (1983), a *transformation* is defined as an equivalence relation between *program schemes* (parameterised templates), where a program scheme represents a class of related programs. Given a program scheme, programs are generated by parameter instantiation. Transformations are partial mappings from one scheme to another such that a domain element and its image under the mapping have equivalent behaviour. They can be represented procedurally, as an algorithm, or declaratively, as a rewrite rule with the domain scheme on the left hand side of a derivation (often $\Rightarrow$) or equivalence (often $\Leftrightarrow$) operator and the range scheme on the right hand side.

An important distinction can be made between *local* and *global* transformation rules. A local transformation rule can be applied to a program P_0 meeting a specification S and in one step yield another program P_1 which also meets specification S. A global rule may require the performance of a sequence of n manipulations to a program Q_0, yielding a sequence of states of the program $Q_1..Q_n$, such that both Q_0 and Q_n satisfy R, but intermediate states $Q_1..Q_{n-1}$ do not. It is often convenient to express local transformations using rewrite rules, and global transformations using procedures (or alternatively *scripts* of syntactic rules).

For convenient discussion, transformational approaches can be placed relative to two orthogonal axes:

- *Transformation in the small* (transformations made locally at the program construct level) vs. *transformation in the large* (transformations made at the architectural level);
- *Horizontal transformations* (notational transformations at one level of abstraction) vs. *vertical transformations* (transformations involving a change in level of abstraction —usually towards the concrete).

Figure 2.6 shows various example transformations relative to these axes:

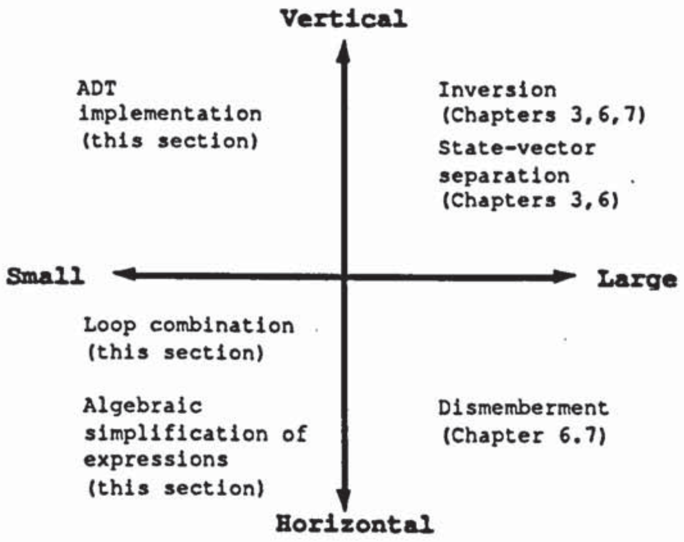

Vertical
ADT implementation (this section)
Inversion (Chapters 3,6,7)
State-vector separation (Chapters 3,6)
Small
Large
Loop combination (this section)
Algebraic simplification of expressions (this section)
Dismemberment (Chapter 6.7)
Horizontal

Figure 2.6. A map of transformational space.

Local rules typically express horizontal transformations in the small. A typical use of such rules is to relate language constructs (Partsch & Steinbrüggen, 1983), for example

```
LOOP: if not B then goto END endif;
S; goto LOOP;
END:

⇔ while B do S endwhile  .
```

This rule can be used either to add or remove structure in a program. Chapter 6 briefly mentions the use in JSD of such a rule for the purpose of circumventing the control-flow limitations imposed by single-entry single-exit blocks (actually, to allow jumps into the body of S). Local rules can also express algebraic properties of programming languages, such as

```
1 + if B then x else y endif
   ⇔ if B then 1 + x else 1 + y endif ,
```

and rules about data-type properties, such as

```
pop(push(s,x)) ⇔ s   -- for unbounded stacks, s and elements, x

b ∧ b ⇔ b            -- for booleans, b  .
```

Global rules typically express vertical transformations in the large, and are often expressed procedurally. The transformations used in JSD implementation fall into this category and are discussed later in the thesis.

There is also an intermediate class of transformations which Partsch & Steinbrüggen (1983) call hybrid rules. The most common of these is the FOLD/UNFOLD pair. UNFOLD is the replacement of a call to a procedure or function by its body, with appropriate substitutions, and FOLD is the inverse transformation, whereby a piece of code is replaced by an equivalent function or procedure call. These two rules are very powerful, and form the basis of the pioneering work on the transformation of functional programs by Burstall & Darlington (1977). UNFOLD is used in Chapter 6 as a component of a global transformation strategy.

Hybrid rules can be used to express programming knowledge about suitable reifications of datastructures. For example, following Cheatham (1984), the rule below provides a linked-list implementation for notation designating the the abstract operation of addition of an element to the end of a queue (here '$' prefixes variables in each program scheme):

```
insert $e into $q
     ⇔  begin
            $q.count := $q.count + 1;
            $q.rear := CreateQueueMember($e,$q.rear)
        end  .
```

This example of a vertical transformation raises several general points about program transformations. Firstly, it is clear that providing the ability to support the reification of queues into another representation (circular buffers, say) would simply be a matter of adding other rules, one for each abstract operation. Secondly, some overall control is needed to ensure that each abstract operation is transformed to be consistent with the reification chosen for a particular implementation. Thirdly, when it is unclear as to which reification of a datastructure is most appropriate (for example, from the perspective of efficiency), alternatives can be generated and tested operationally. Fourthly, different reifications can be chosen to custom tailor (Cheatham, 1984) and thus re-use a given specification for different environments.

Transformation of Functional Programs

Functional programs are easier to manipulate than imperative notations, having strong algebraic properties (Henderson, 1986). Much work in the area of transformations has therefore focused on functional languages (Darlington, 1982). Recognising that it will often be desirable to produce final implementations in imperative languages running on conventional hardware, much of the effort in transformational programming has been on taking abstract inefficient functional programs (which can be regarded as executable specifications) and transforming them into a form to facilitate easy final translation into efficient imperative implementations (Partsch, 1991). This often involves the introduction of imperative idioms into a program through strategies such as loop combination, the disciplined introduction of assignment and other destructive operations, and recursion removal. Some work, such as the Munich CIP project (Broy & Wirsing, 1991), has used *broad spectrum* languages which incorporate both applicative and imperative notations within one language. Other approaches have explicitly translated from a functional language to an imperative one. For example, Boyle & Muralidharan (1984), discuss a system for transforming Lisp specifications into Fortran programs.

Darlington (1982) gives six correctness-preserving transformation rules for rewriting functional programs:

(i) **Definition.** Introduce a new function.

(ii) **Instantiation.** Introduce an instance of an existing equation with one or more of its variables instantiated consistently.

(iii) **Unfold.**

(iv) **Fold.**

(v) **Abstraction.** Introduce a where clause by deriving from a previous equation `E = E'` a new equation

$$E = E'[u_1/F_1, \ldots u_n/F_n] \quad \textbf{where } \langle u_1, \ldots, u_n\rangle = \langle F_1, \ldots F_n\rangle$$

`E[E1/E2]` means `E` with all occurrences of subexpression `E2` replaced by `E1`.

Angle brackets (<   >) form a tuple.

(iv) **Laws.** Use algebraic laws (associativity, commutativity, etc) on the right-hand side of an equation to obtain a new equation.

Darlington gives several examples illustrating the application of these rules to NPL programs. As a straightforward example of the approach, consider the loop combination transformation. Functional programs tend to have many independent computations kept apart for clarity. Often these will traverse the same datastructure once for each computation. Loop combination rearranges the program so that each computation is performed during a single traversal. Consider the following NPL recursion equations (in which lists have two constructors: `nil` and 'cons', denoted by the infix operator '`::`'):

```
-- add all elements of a list
sum(nil) = 0
sum(n::l) = n + sum(l)

-- multiply all elements of a list
prod(nil) = 1
prod(n::l) = n * prod(l)   .
```

A function `g` may be specified as

```
g(l) = sum(l) + prod(l)  .
```

This involves two traversals of the list `l`. To combine the loops, Darlington first defines a new equation

```
k(l) = <sum(l),prod(l)>
```

and by abstraction and folding obtains

```
g(l) = u + v where <u,v> = k(l)  .
```

From here transformation proceeds as follows:

```
k(nil)    = <sum(nil),prod(nil)>      -- instantiation of list to
nil
          = <0,1>                     -- unfold (defn of sum and
prod)

k(n::l)   = <sum(n::l),prod(n::l)>  -- instantiation of list to
n::l
          = <n+sum(l), n*prod(l)>     -- unfold (defn of sum and
prod)
          = <n+u, n*v>
                where <u,v> = <sum(l),prod(l)>
                                      -- abstraction
          = <n+u,n*v)
                where <u,v> = k(l)   -- fold (defn of k)
```

Collecting the derived equations together gives the following definition of `g`:

```
k(nil) = <0,1>
k(n::l) = <n+u,n*v)
          where <u,v> = k(l) .
```

This definition is more efficient that the original one. It is also more difficult to understand. This example clearly shows the basic utility of the transformational approach. Specifications are written with clarity as the goal. They are then converted into an efficient form by correctness-preserving steps. Balzer (1981) gives another discussion of this style of transformation and gives an example of transformational implementation of a formal specification for the 8-Queens problem into a Lisp implementation.

A Formal Transformational Paradigm

Partsch (1991) presents a fully formal transformational approach to software development, which although heavily based on the CIP project (Broy & Wirsing, 1991) also serves as a general account of the current state of transformational programming methodology. The development process espoused by Partsch can be summarised as follows:

- (descriptive) formal problem specification;
- functional (non-deterministic, recursive) solution;
- deterministic, tail-recursive solution;
- further modified applicative program (with loop combinations, shared datastructures, etc);
- efficient procedural program (perhaps with recursion removed).

States are optional and the formal process may start or end at any stage. All transformations are performed by the application of transformation rules whose correctness has been proved correct. The CIP approach assumes that the developer will choose transformations which will then be applied by the software tool. Boyle & Muralidharan (1984) discuss a transformation system for LISP-to-Fortran conversion, which while sitting comfortably in the process model given above, automatically selects and applies suitable transformations.

Transformation-in-the-large

Jackson (1983) describes the use of program transformations for the implementation of the highly concurrent networks represented by a JSD specification. The transformations he describes are very different from those of Darlington, however, in that they are essentially transformations in the large, i.e. they work at the module level. The purposes of transformations in JSD are the following:

- reduction of concurrency
- replacement of asynchronous communication with synchronous communication
- imposition of a scheduling discipline on asynchronous communicating processes
- implementation of multiple instances of a process by allowing them to share re-entrant programs
- implementation of persistent storage of process state information
- reduction of indeterminacy.

These objectives are likely to be important in the implementation not only of JSD specifications, but also of specifications produced by any other operational method employing asynchronous parallelism. The present work focuses on JSD in particular, as it has a well developed repertoire of transformations for achieving these objectives (the major effort in the PAISLey project, on the other hand, appears to have been in facilitating the direct execution of specifications rather than their transformation (Zave, 1991)).

The correctness-preservation of the standard Jackson transformations is generally argued on a pragmatic basis (Jackson, 1983; Cameron, 1989). However, there is potential for a fully formal approach to JSD transformation. Hoare and Shridar (1985) re-express some of Jackson's examples in CSP and show how the algebraic properties of the notation can be used to reduce concurrency and Yeung *et al.* (1991) further develop this approach. Goldsmith (1988) describes an occam transformation system (OTS). Given the relationship between occam and CSP, and CSP and JSD, one could envisage the OTS's use to perform Jackson-like architectural transformations in an algebraic style.

This research is concerned with the implementation of a class of JSD transformation known as *dismemberment*. As of Summer 1992, no published method of automating dismemberment has appeared (although see Section 1.3 on Related Work). Dismemberment is an example of horizontal transformation-in-the-large. Processes are split into smaller processes which each implement a subset of the original's behaviour. This allows the implementor greater flexibility of scheduling and distribution of processing than would be available otherwise. As it is possible to formalise JSD processes in terms of regular languages (see Chapter 3), and as it is straightforward to talk about subsets of formal languages (and corresponding partitions of automata), the approach taken has been to exploit existing knowledge about that formal language theory to provide a (procedural) method of implementation for the dismemberment transformation.

2.4 Concluding Remarks

This chapter has argued that the conventional approach to software development — seen fundamentally as a combination of TDFD and the Waterfall process model — is flawed, as it fails to take into account the evolutionary nature of the software development context of most substantial projects. The operational paradigm, as espoused by Zave (1982) and others, appears to offer a promising alternative. This approach proceeds by constructing an executable model of the user's problem domain, elaborates the model to produce the desired functional behaviour, and implements the specification by automatic transformation. Ratcliff (1990) summarises the advantages of the approach as follows:

- transformational implementation can be automated;
- 'maintenance' becomes a specification level activity — modifications are made to a problem-oriented artifact which is then reimplemented automatically;
- prototyping can be supported, as objects with executable semantics appear early in the software process;
- specifications can be reused by transforming them into different implementations.

Despite these strong arguments in its favour, the operational paradigm has not, as yet, achieved the popularity hoped for by its pioneers in the early and mid 1980s. Both researchers and practitioners have tended to favour evolutionary changes to development methodology and when faced with operational methods, have sometimes failed to grasp their revolutionary nature (Zave, 1991b); one recent paper, for example, criticises the method for not supporting top-down development (Hull, O'Donogue & Hagan, 1991).

Boehm (1988), while making no major methodological challenge to what he terms the 'transform' model, does identify the following practical difficulties:

- limited availability of automated support;
- possible difficulties in a transformationally derived system accommodating unplanned evolution;
- potential problems in keeping track of, and making good choices among, the ever-increasing supply of reusable components and commercially available software products which may feature in target environments.

Ratcliff (1990) shows how the PRESTIGE JSD transformational workbench (Bass *et al.*, 1991) addresses the first two of these reservations and partially mitigates the third. Firstly, PRESTIGE offers general support for JSD, a method with wide application in both real-time and information system domains. Secondly, the workbench allows alteration of all the elements of a JSD specification which may be subject to evolution. Thirdly, the architecture of the PRESTIGE system, which performs as much of the transformational process as possible in implementation-independent terms, should reduce turbulence felt when PRESTIGE-generated implementations are targeted at new environments. The PRESTIGE system is discussed in more detail in Chapter 7.

Chapter 3
On the Relationship of JSD to Formal Language Theory

3.1 Introduction

This chapter relates JSD to concepts from formal language and automata theory. Since the latter two areas are intimately connected (Hopcroft & Ullman, 1969), their names will often be used interchangeably. Language theory is a fundamental part of computer science and either explicitly or implicitly underlies many of the activities of computer scientists and software engineers. Figure 3.1 presents a view of relationships among various subdomains in computer science. In particular, it is intended to imply an inheritance-like structure in which both JSD and compiling are seen as subordinate to language theory. This chapter will concentrate on the validity of viewing JSD in this way. The position of compiling is well supported by any number of sources (for example, Aho *et al.*, 1985; Backhouse, 1979).

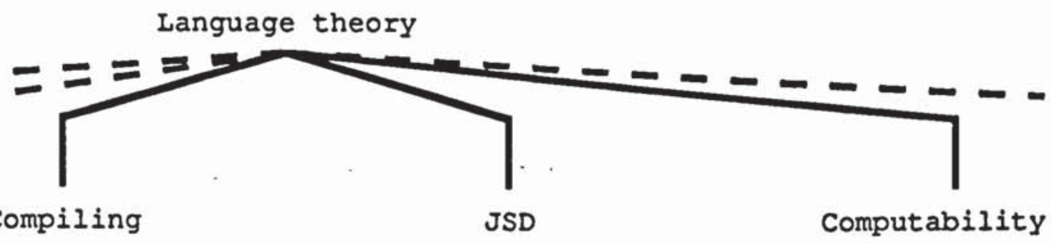


Figure 3.1. Inheritance relationships among some subdomains of computer science.

Often when composing an inheritance graph, some attribution will be made to a fairly lowly node which actually has greater generality than is implied by its position in the hierarchy. Once this generality is noticed, the attribute can be shifted upwards to a parent node thus allowing other child nodes to benefit from this characteristic by inheritance. The present work involves such a reorganisation to demonstrate the applicability of algorithms developed in the area of compiling to JSD implementation. This is illustrated in Figure 3.2.

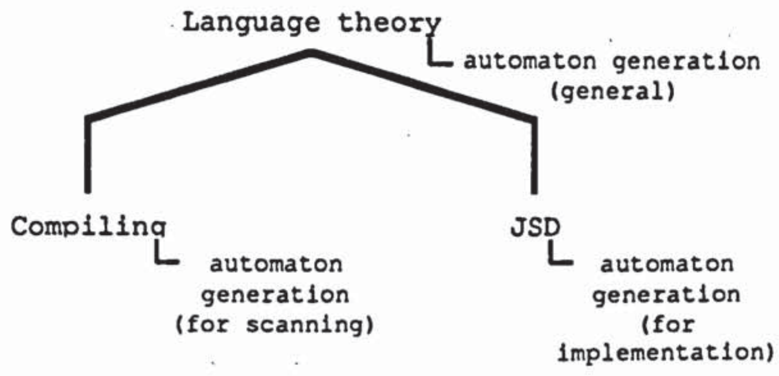

Language theory
automaton generation (general)
Compiling
automaton generation (for scanning)
JSD
automaton generation (for implementation)

Figure 3.2. Generalising ideas from compiling so as to apply them to JSD.

According to Dwyer (1991), the link with formal language theory was always well known to the Michael Jackson-led programming team which developed JSP. It was never emphasised on official JSP and JSD courses because the methods were originally aimed at a data processing audience. Hughes (1979) provides the first published account of any aspect of the Jackson methods in these terms — she deals with the program design method JSP. Jackson (1983) shows how rules for the transformation of regular expressions (Minsky, 1967) can be used to manipulate entity structures and introduces the application of *follow sets*, a property of formal grammars (see Backhouse, 1979, for example), for systematic error detection. The possibility of regarding JSD specifications as networks of communicating finite automata is recognised, though not developed, by Borgers and Munro (1990), while Zave (1985) discusses similar networks as a medium for expressing operational specifications.

3.2 The Modelling Phase of JSD

A JSD specification of a system is based on an abstract model of a real world problem domain (Jackson, 1983; Cameron, 1988; Renold, 1988b). The activity of modelling involves abstracting important features from the problem domain; information is lost, but at the gain of a more tractable view of the problem. The modelling process therefore functions as a method of controlling complexity. There is a semantic gap between a real world domain and a JSD model of the domain and this gap reflects the amount of information lost in the abstraction process. The description of JSD presented here is still more rarified and models particular features of JSD specifications in terms of language and automata theory. The advantage of modelling JSD at this more abstract level is that it allows results from the abstract theories to underpin the design and implementation of new transformation strategies. A similar method of investigation is adopted by Sridhar & Hoare (1985) who model JSD in terms of the more abstract CSP notation, and by doing so are able to relate JSD to CSP's formal semantics (Hoare, 1983). They go on to show how the algebraic laws governing CSP operators could form the basis of an algebraic approach to JSD transformation.

The following discussion relates concepts from JSD, language and automata theory, set theory and for further illustration CSP. Table 3.1 summarises analogous terminology across these domains.

JSD	Language theory	Automata theory	CSP	Set theory
action	letter/symbol	symbol	event	element
action list	alphabet	alphabet	alphabet	set
life history	string	string	trace	sequence
structure diagram	regular expression	state transition graph	process description	set of sequence
process	language	automaton	process	—

Table 3.1. Analogous terminology across a variety of domains

The Vocabulary of a JSD Specification

Any object in an operational specification must have an unambiguous meaning in real-world terms if the specification is to be held to reflect accurately the problem domain. A great advantage of JSD's approach to modelling is that the bridge between the rich informality of the real world and the formality of an executable JSD specification is very narrow, and is localised to a dictionary of *event* descriptions (Jackson, 1988). This is the only place where specification elements are described in an informal way. The meaning of all other terminology used in a particular specification can be infered, either directly or indirectly, from these event descriptions. The starting point in the construction of a JSD domain model is therefore the identification and description of important events which occur in the real world of interest (Jackson, 1983; Cameron, 1986).

A JSD *action* is an abstraction of some real world event which can be regarded as atomic for the purposes of system development†. Each action is described informally, in the jargon of the user, and the set of actions then comprises a vocabulary of terms to be used in describing the system. An example of a list of action definitions from a simple banking example is given below.

open	An account is opened
credit	Money is deposited in an account
debit	Money is withdrawn from an account

† Broy (1991) makes the useful distinction between events — properties of the real world — and actions — properties of a model. This distinction is adopted in the thesis.

`close`   An account is closed

The developer may also associate a tentative list of *attributes* with each action (for example, `open` may be assigned attributes `account-number, customer, date`) which may be elaborated in the light of functional requirements; action attributes are discussed later. The action list defines a vocabulary of the terms in which the system and its functions will be discussed. This list of actions therefore starts to define the scope of the system, which can only produce output based on the detection of the corresponding real-world events (Cameron, 1986; Renold, 1988b).

The modelling of individual events provides the basis for the development of a specification. The JSD developer then goes on to model domain *behaviour* as sequences of events. Language theory (Hopcroft & Ullman, 1969; Rayward-Smith, 1983) provides convenient ways to discuss this modelling technique.

Necessary Fundamentals of Language Theory

An alphabet, A, is a finite set of *symbols* (or *letters*). Given an alphabet, it is possible to generate *sequences* of letters called *strings*. A string of length k is a member of $A^k = A \times A \times \ldots \times A$, which is the cartesian product of A with itself k times. The set of non-empty strings over A is defined as

$$A^+ = \bigcup_{n=1}^{\infty} A^n$$

and is called the *transitive closure* of A (Backhouse, 1979). Let ε be the empty string. Then the set containing just ε is A^0. We can now define the set of all strings of alphabet A as

$$A^* = \bigcup_{n=0}^{\infty} A^n \quad .$$

A^* is referred to as the *reflexive transitive closure* of A. It is also called the *Kleene closure.*

Strings are usually denoted by the juxtaposition of their elements (e.g., `aabbbcdaaaaa` is a string over the alphabet `{a,b,c,d}`). As most of the symbols referred to in the thesis will themselves be made up of sequences of letters and numbers, it will be useful to denote alphabets and strings using explicit separators and delimiters. It is convenient to use CSP syntax (Hoare, 1983; Sridhar & Hoare, 1985). The alphabet of a language LANG is denoted αLANG. Strings are delimited by angle brackets, with individual elements separated by commas. <> denotes the empty string ε. Here are some examples of alphabets together with some example strings:

```
αMONEY = {£,1,0}
<£,1,0,0,0,0,> ∈ αMONEY*

αENGLISH = { l | l ∈ a..z}
<c,o,n,c,a,t,e,n,a,t,i,o,n> ∈ αENGLISH*

αACCOUNT = {open, close, credit, debit}
<open, debit, debit, credit, debit, close> ∈ αACCOUNT*  .
```

Various operations can be performed on strings. Concatenation is denoted by the ^ operator; for example

```
<open, debit>^<credit> = <open, debit, credit>  .
```

$s\lceil A$ is the string s restricted to the elements in set A; for example

```
<open, debit, debit, credit, debit, credit, close>⌈{open, close,
credit}
      = <open, credit, credit, close>  .
```

The first item in a non-empty string is denoted by s_0 while the rest of the string is denoted by s'; for example

```
<open, credit, close>₀ = open
<open, credit, close>' = <credit, close>  .
```

$s \leq t$ means that s is a *prefix* (initial subsequences) of t; for example

```
<open, credit> ≤ <open, credit, close>  .
```

The empty string is the prefix of all strings.

Syntactic Structure

The Kleene closure T* represents all possible sequences of letters in an alphabet T. We are usually more interested in particular subsets of T* which exhibit some structure. Such subsets are called *formal languages* (Rayward-Smith, 1983; Backhouse, 1979). This section shows two equivalent approaches to language definition and relates them to JSD *structure diagrams*.

Formal Languages

A formal language over an alphabet T is a subset of T*. The usual way to define a formal language is by providing a *grammar* for that language. A grammar is defined by a 4-tuple G = (N,T,P,S) where

- N is a finite set of non-terminal symbols.

- T is a finite set of terminal symbols.
- $S \in N$ is a distinguished symbol called the start symbol.
- P is a set of productions each of which has the form *lhs* → *rhs* where *lhs* $\in (N \cup T)^+$ and *rhs* $\in (N \cup T)^*$. *lhs* is called the left-hand side and *rhs* the right-hand side of the production (Backhouse, 1979.).

Chomsky (1957) arranges grammars into a hierarchy, numbered from 0 to 3, according to constraints on the form of productions. Type 0 is the least restricted form, while Type 3, the regular grammar, is the most restricted. Consideration of grammars is here focused on those of type 3 (type 2 grammars, the so-called *context-free* grammars are briefly discussed at the end of the chapter). A regular grammar is a grammar $G = (N,T,P,S)$ in which, either

- all productions have the form $A \rightarrow tB$ or $A \rightarrow t$ where t is a terminal (i.e. $t \in T^*$) and A and B are non-terminal symbols (i.e.$A, B \in N$). This form is called a right-linear grammar, or

- all productions share the form $A \rightarrow Bt$ or $A \rightarrow t$ where $t \in T^*$ and $A,B \in N$. This form is called a left-linear grammar (Backhouse, 1979.).

Grammars in which left-linear productions are intermixed with right-linear productions are not regular. For example, the grammar having the productions

```
S →   ( R
S →   ε
R →   S )
```

is not regular. This grammar represents strings of balanced parentheses $\{ "("^n \; ")"^n \mid n \geq 0 \}$, and is discussed further at the end of the chapter.

Consider the following right-linear grammar G = ({ACCOUNT,TRANSACT}, {open,credit, debit,close},P,ACCOUNT), where P consists of

```
ACCOUNT  →  open TRANSACT
TRANSACT →  credit TRANSACT
TRANSACT →  debit TRANSACT
TRANSACT →  close    .
```

The language described by this grammar (a subset of {open,credit,debit,close}*) includes the following strings:

```
<open, close>
<open, debit, credit, close>
<open, credit, credit, debit, credit, close>   .
```

It can be shown how ACCOUNT *generates* these and other strings by considering the productions as a set of rewrite rules. Following Backhouse, the $\Rightarrow$ (derivation) operator is used to denote an instantiation of a production. For example,

```
ACCOUNT  ⇒ open TRANSACT
         ⇒ open credit TRANSACT
         ⇒ open credit debit TRANSACT
         etc.
```

A chain of derivations $x_1 \Rightarrow x_2 \Rightarrow x_3 \Rightarrow ... x_n$ is called a *derivation sequence* (Backhouse, 1979.). In this case, the progressively unfolding derivation sequence can be considered to model a possible behaviour of a bank account (abeit at a rarified level of abstraction).

Regular Expressions

Regular expressions are an elegant notation for regular languages (Rayward-Smith, 1983). The regular expressions over an alphabet A are defined recursively as follows:

1. $\emptyset$ denotes the empty set
2. ε denotes $\{\varepsilon\}$
3. a where $a \in A$ is a regular expression denoting $\{a\}$
4. If r_1 and r_2 are regular expressions, representing languages L_1 and L_2 respectively then

   a) $(r_1 \mid r_2)$ (representing $L_1 \cup L_2$)
   b) $(r_1 \cdot r_2)$ (representing $L_1 \cdot L_2$)
   c) $(r_1 *)$ (representing L_1*)

   are all regular expressions.
5. Nothing else is a regular expression.

The following regular expression describes the same language as the regular grammar for ACCOUNT above:

```
open • ( (credit | debit)*) • close  .
```

Brackets are often omitted according to operator precedence in the order $*$, $\cdot$, $\mid$. It is also common to omit $\cdot$ and denote it by juxtaposition. This allows the example above to be abbreviated as follows:

```
open (credit | debit)* close  .
```

Regular expressions are generally considered to be easier to interpret than equivalent grammars and therefore tend to be favoured for the description of regular languages

(Backhouse, 1979.). Common uses of regular expressions in computing include the definition of lexical items in programming languages, and as parameters to string pattern matchers such as the UNIX *grep* ("get regular expression") command (Aho *et al.*, 1985).

Entities and Event Orderings

Having defined the vocabulary of a system, the JSD developer then goes on to identify and model structure in the traces of events in the real world. Such a model increases understanding of the dynamics of the real world and provides the basis for the behaviour of the system under development (Renold, 1988b).

The action abstraction in JSD models real world events as atomic (Jackson, 1983). For the purposes of the following discussion, it is assumed that at any discrete time `t` only one action can occur, so that the behaviour of the real world can be regarded as a single stream of events (the issue of simultaneous unconnected events in the real world is resolved by regarding their relative ordering as unimportant). This single event stream view of the real world is not very informative, so JSD leads the developer to find and represent regularities in the stream (Jackson, 1988). One way is to note pairwise ordering constraints on events (for example, in modelling a bank account, we notice that `open` always appears before `close`). In a problem for which JSD is applicable, this process leads to the discovery of *independent* traces of events interleaved in the event stream. Examination of these traces will often reveal *entities* in the world which suffer or perform the actions which make up the traces. These entities exhibit their behaviours concurrently and asynchronously (there may actually be some synchronisation, discussed shortly). The various ordering constraints on these traces can be represented using regular expressions or regular grammars (Jackson, 1988). In practice, an equivalent diagrammatic notation for regular sets, called the *structure diagram*, is employed. Figure 3.3, after Hughes (1979), shows the correspondence between regular expressions and structure diagrams.

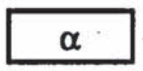
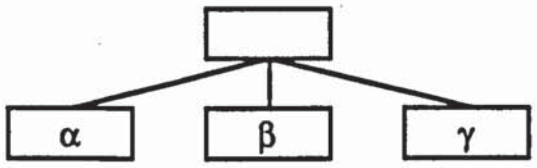
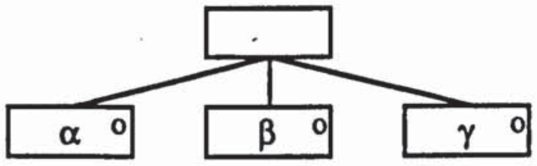
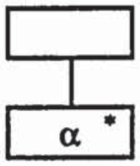
α	
$(\alpha.\beta.\gamma)$	
$(\alpha \mid \beta \mid \gamma)$	
$(\alpha)^*$	

Figure 3.3. Equivalence of regular expressions with Jackson trees, after Hughes (1979).

Figures 3.4 & 3.5 show examples of structure diagrams, together with equivalent regular expressions.

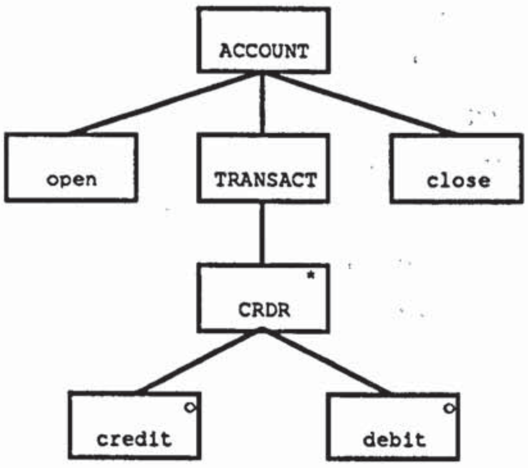


Figure 3.4. Bank `ACCOUNT: open (credit | debit)* close.`

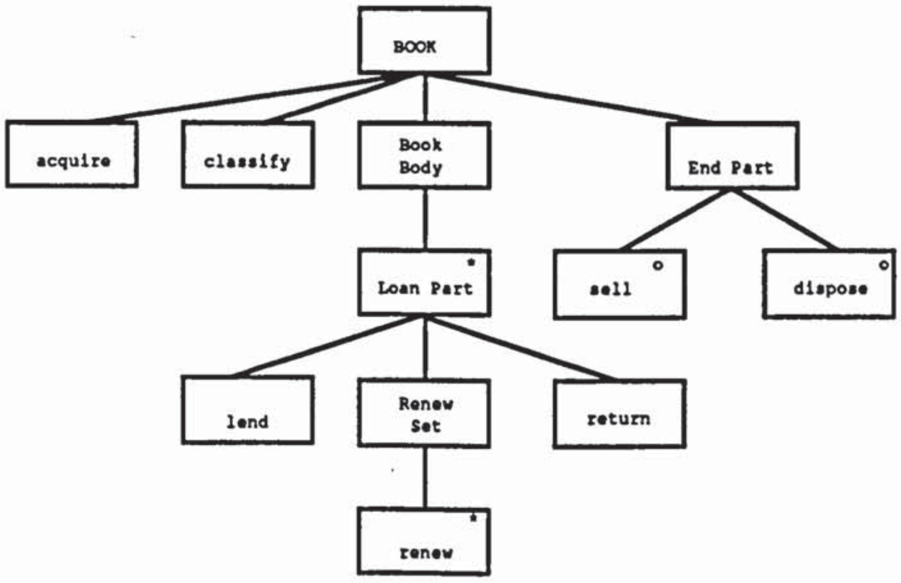


Figure 3.5. Library book: (Cameron, 1986).

```
acquire classify (lend renew * return ) * (sell | dispose)) .
```

The ability to discover members of regular sets of event traces (regular languages) in the real world is one way to discover the applicability of JSD to a particular problem.

The definition of the trace semantics of the CSP operator || (parallel composition), elegantly expresses the relationship between the single real world event stream and the traces of individual entity behaviours. Jackson (1983) and Cameron (1986) acknowledge the influence of CSP on JSD; Sridhar & Hoare (1985) and Yeung *et al.* (1991) further examine the relationship. In CSP, the set of all possible behaviours which a process P can perform is denoted `traces(P)`. (This is equivalent to the language denoted by a regular expression with the same structure as P). P || Q is the parallel combination of two processes P and Q. Assuming for the purposes of illustration that the only events observable in the real world are those participated in by the entities modelled by P and Q, then the real world event stream is a member of the set traces (P || Q). The relationship between `traces(P || Q)` and the individual traces of the participating processes is expressed by the definition of the parallel composition operator (Hoare, 1983):

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q$$
$$\mathrm{traces}(P \parallel Q) = \{ s \mid s \in (\alpha P \cup \alpha Q)^* \wedge s \restriction \alpha P \in \mathrm{traces}(P) \wedge s \restriction \alpha Q \in \mathrm{traces}(Q) \}$$

Events common to the alphabets of P and Q must be participated in by both processes simultaneously. Each process is free to perform the other events in its alphabet independently. One way to see the initial stages of JSD modelling is as a process of identifying alphabets and trace structures such that the definition of || holds.

Having identified trace structures, the next step is to relate them to entities which can be considered to perform or suffer behaviour according to the ordering constraints imposed. It is not always straightforward to relate trace structures to particular entities, and it is necessary to consider the following cases:

- interleaved traces specified by the same structure;
- different structures seemingly related to the same entity;
- events seeming to take part in more than one trace (of different entities)

These three cases are discussed in more detail below.

Interleaved Traces from the Same Regular Set

Interleaved traces specified by the same structure indicate multiple instances of the same entity class. This is known as a multithreaded structure clash in JSP (Jackson, 1975), and more generally as a multiplexing/demultiplexing problem. Consider the following example:

$$s, t \in \mathtt{traces(ACCOUNT)}$$
$$s = \langle open_s,\ debit_s,\ close_s \rangle$$
$$t = \langle open_t,\ credit_t,\ credit_t,\ credit_t,\ debit_t,\ close_t \rangle$$

$$realworld = \langle open_s,\ debit_s,\ open_t,\ close_s,\ credit_t,\ credit_t,\ credit_t,\ debit_t,\ close_t \rangle\ .$$

The distinction between those actions participating in trace s and those in trace t is made by reference to attributes associated with each action which uniquely identify the entity of whose life-history they are part.

Different Structures Related to a Sngle Entity

Sometimes it will seem natural to associate two or more structures with the same entity (Cameron, 1986). Efforts to unify the structures will be prevented by the observation of arbitrary interleaving of instances of the trace sets. Such structures are said to represent distinct *roles* of the entity. Consider the simple example of an EMPLOYEE who clocks on and off work, and performs jobs which are long enough that they cannot always be finished in a single day.

```
αEMPLOYEE = {clock_on, clock_off, start_job, end_job} .
```

The following activity may be observed in the real world:

```
realworld =
   <clock_on,start_job,end_job,start_job,clock_off,clock_on,
      clock_off,clock_on,end_job,clock_off>.
```

While the members of `{clock_on,clock_off}` are constrained relative to each other, as are members of `{start_job,end_job}`, it is impossible to impose a relative ordering across these subsets of αEMPLOYEE. The behaviour of EMPLOYEE can be appropriately represented as two roles, EMPLOYEE_DAYS and EMPLOYEE_JOBS which have the structures

```
    (clock_on clock_off)* &
    (start_job end_job)*,
```

respectively.

Events Shared by Traces of More Than One Entity

It is sometimes appropriate to resolve the stream of events into traces by regarding some actions as participating simultaneously in more than one trace. Such actions, called *common actions*, represent points of synchronisation in the life-histories of entities. For example, in a missile defence system, we may regard the impact of a defending missile with an incoming target as a single event, and represent that event as an action appearing in the life histories of both entities. The choice of using common actions is up to the developer (Jackson, 1983) and in the end comes down to a matter of style.

Remarks Concerning Entity Modelling

It has been suggested here, following Jackson (1988), that entities are discovered through an attempt to explain observed regularities in the occurrence of events. This is only one perspective on JSD modelling and discovering entities is not always as neat as it may appear from the discussion above. In practice, there will almost certainly be a need to *impose* some regularity as part of the abstraction process. The approach presented here has been chosen because it compliments a language theoretic view of JSD, but Jackson (1983) suggests other, less formal, ways of discovering actions and entities (for example, by examining a textual description of the real world for verbs and nouns). It is likely that a combined approach to entity identification and modelling will be the most effective for non-trivial problems.

Machines

A key feature of a JSD structure diagram is that it can be given an operational interpretation. The most straightforward way to do this is to convert the structure diagram into a program according to the following rules (based on Jackson, 1983):

- sequences become compound statements;
- selections become conditional statements;
- iterations become iterative statements;
- leaf nodes become skip statements which act as place markers for possible executable operations;
- single read-ahead of actions (explained below) is applied.

Read statements are placed in the text of the program so as to accept messages representing each of the actions which mark the leaf nodes. This *standard read-ahead* technique (Jackson, 1975) is achieved by inserting a read at the beginning of the program and then immediately *after* each skip statement (except the last). The boolean expressions required by conditional and iterative statements are then merely predicates on the value of the most recently read action. The bank ACCOUNT structure is realised as the following program:

```
ACCOUNT seq
   read(input);
   skip; -- open
   read(input);
   transact itr while input ≠ close
      crdr sel input = credit
         skip; -- credit
         read(input);
      crdr alt input = debit
         skip; -- debit
         read(input);
      crdr end;
   transact end;
   skip; --close
   -- a read here would be redundant
ACCOUNT end.
```

Each program produced by the read-ahead technique can be regarded as an abstract 'long-running' sequential *process*. The term 'long-running' is used to demonstrate that a program executes once in correspondence with the entire life of an entity. An executing JSD specification can be regarded as a set of such processes running concurrently and asynchronously (except where they share common actions). The animated specification takes as input a stream of messages denoting the real-world event trace and simulates the behaviour of the modelled entities in response to these events.

As the specification is elaborated, the skip statements can be replaced with system-state update and output statements of various sorts. The constraints provided by the structure diagrams ensure that updates to the state of the system can occur only in a fashion consistent with changes in the real world. Providing that an inevitable time delay along the links between the world and the system is accepted, the model will reflect the state of the real world at any moment in time and will thus provide the basis for the functionality of the system. Note that this implementation scheme assumes error free input. More is said about this issue in Chapter 6.

Finite Automata

The read-ahead technique preserves the structure of a process specification in the behaviour-producing mechanism. It is not necessary to insist on this however, and a particularly interesting alternative is to transform structure diagrams into *finite automata* (FAs). These can then support the generation of alternative program structures which are behaviourally equivalent to those resulting from a read-ahead approach. In some cases, to be discussed at length in the rest of the thesis, these alternative structures can offer considerable advantages.

Automata theory (Minsky, 1967; Hopcroft & Ullman, 1969, for example) is concerned with the study of abstract computing devices. It is closely related to language theory (Hopcroft & Ullman in particular stress this link) and provides a useful way to characterise the semantics of JSD models, because it fits in with the view of the operational approach discussed in the last chapter in that it provides a way of discussing the internal structure of specification elements without biasing the way such a specification could be implemented.

A deterministic finite state automaton (DFA) is a mathematical model defined by the 5-tuple $(S, \alpha M, \delta, F, q_1)$ where:

- S is a finite non-empty set of *states*
- αM is an alphabet of inputs
- δ is the *transition function* to map $S \times \alpha M \rightarrow S$
- F is a finite set of *accepting* (or *final*) *states*
- q_1 is a distinguished element of S (the *start state*).

The following automaton models a bank ACCOUNT:

```
S = {1, 2, 3},
αACCOUNT = {open, debit, credit, close},
δ = {(1, open, 2), (2, debit, 2), (2, credit, 2), (2, close, 3)},
F = 3,
q1 = 1 .
```

Figure 3.6 shows a representation of this machine in the form of a *transition graph* where nodes represent states, and labelled arcs represent transitions on particular inputs. Accepting states are represented by nodes with doubled boundaries.

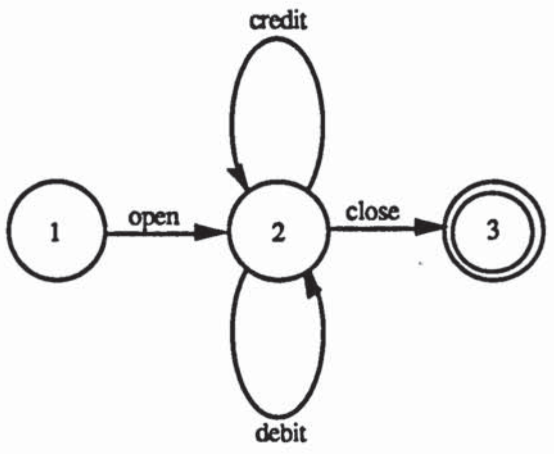


Figure 3.6. A transition diagram for ACCOUNT.

Any string of inputs which takes a machine M from its start state to one of its accepting states can be said to have been *recognised* or *accepted* by M. The set of strings accepted by M is called the *language* accepted by M, and may be denoted L(M). L(M) is defined as the set $\{x \in \alpha M^* \mid$ accepts(M,x)$\}$. This notion of acceptance can be expressed more formally as follows. The transition function δ: $S \times \alpha M \rightarrow S$ can be extended to include its transitive closure (Rayward-Smith, 1991) to δ: $S \times \alpha M^* \rightarrow S$ by the following definition:

$$\delta(S, \varepsilon) = S$$
$$\delta(S, ax) = \delta(\delta(S,a), x) \qquad [\text{where } a \in \alpha M \wedge x \in \alpha M^*].$$

L(M) is then defined:

$$L(M) = \{x \mid \delta(q_1, x) \in F\} .$$

Given a DFA we can simulate its behaviour according to the following algorithm (Rayward-Smith, 1991):

```
state := $q_1$;
while not end of input do
   read(input);
   state := $\delta$(state, input);
endwhile;
if state $\in$ F
   then write(1)
   else write(0);
```

This provides an implementation of a function fACCOUNT: αACCOUNT* $\rightarrow$ {1,0}. The program will return 1 if it accepts the input (i.e., the input is a member of L(ACCOUNT)), and a 0 otherwise. A language is regular if and only if it is accepted by a DFA (Minsky, 1967).

Nondeterministic Finite Automata

Non-deterministic automata (NFAs) differ from DFAs in two ways:

- the transition function delivers a *set* of states;
- some transitions, called ε-transitions, can be made without consuming input.

Figure 3.7 shows a possible NFA for the regular expression `(a b | a c)*`. Note that this example has been constructed purely to illustrate the distinguishing features of NFAs.

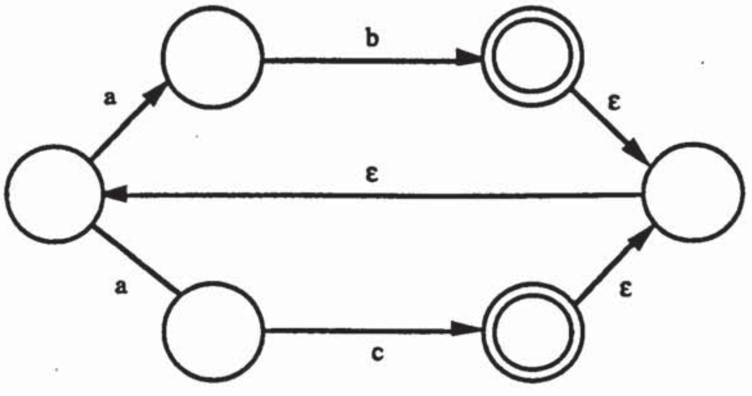


Figure 3.8. A possible NFA for the regular expression `(a b | a c)*`

Implementation of NFAs is more involved than for DFAs; backtracking is required to deal with the presence of alternative transitions for some pairs of states and inputs; in the presence of an alternative, an arbitrary choice must be made. Should a choice lead to a nonaccepting path, it will be necessary to backtrack to the choice point and try another alternative.

The Meaning of State in a Finite Automaton

The history of a finite automaton M is a trace of all the events which it has accepted up to the present. Clearly, as M's behaviour is only determined by its inputs, M's history must be the sole determiner of M's current internal configuration. As long as there is scope for iteration, the set of all possible traces αM^* is infinite; however, M itself is finite — it cannot have a unique internal configuration to represent the result of accepting each of its possible histories. It can be said that M cannot *distinguish* each of its possible behaviours. In fact,

> "[a] machine can distinguish, by its present and future behaviour, between only some finite number of classes of possible histories. These classes..[are]..the 'internal states' of the machine" (Minsky, 1967).

So, a state of M is a label given to an equivalence class of histories of M. The members of such a set of histories are equivalent in the sense that subsequent behaviour of M will be the same following any of the equivalent histories.

By modelling life histories, the JSD developer is abstracting out differences in the behaviour of real world entities in order to construct a tractable model of reality. The nature of the loss of information implied by the finite automaton abstraction was noted by McCulloch & Pitts (1943) in their landmark paper on neural networks:

- the use of disjunction means that a previous state cannot be completely determined from the present state;
- the use of iteration means that it is impossible to determine the time in the past when the first event in the history occurred.

When the information lost is important, because it is required to support system functions, it can be stored explicitly either in a local database called (confusingly, given the present topic) the process's *state vector*, or in a buffer called a *datastream*.

Automata can be used to simulate the abstract behaviour of the entities being modelled — each real world entity is shadowed by one or more corresponding processes (executing automata) concerned with the recognition of a trace of events as a valid life-history of the entity in one of its roles. This provides an alternative, less implementation-oriented interpretation of the semantics of a JSD model than the standard read-ahead based technique described earlier.

3.3 The Network Phase of JSD

The second phase of JSD development is concerned with the elaboration of a model into a network of communicating processes. Some of these processes, the model processes, shadow real world entities and keep track of their behaviour, while others, function processes, validate the inputs to the system, produce its outputs, and perhaps also generate simulated events which are fed back into the model (Jackson, 1983; Cameron, 1986; Renold, 1988b).

Communication

There are two primitive *interprocess communication* (IPC) mechanisms in JSD, introduced in the previous chapter. Only one of them, the datastream connection, is relevant to the event model. The other, state vector inspection is an observer operation which takes place without directly influencing the behaviour of either the inspecting or inspected process, and as such it can be considered as independent of the event model. Consideration of controlled and conversational datastreams (Renold, 1988b) is outside the scope of this work.

Datastreams are idealised first-in first-out buffers used to pipe data from one process to another. They can be regarded as potentially infinite streams of messages, and provide the input and output environment of the model processes represented by the executing automata produced in the modelling phase. A model process will patiently work its way through its input stream, and from time to time may produce outputs for consumption by other processes.

The characterisation of JSD processes as executing finite automata formalises the notion of a process's state as an abstraction of its execution history. It has been seen that modelling entity behaviour in this fashion causes information about execution history to be lost. Where necessary, JSD allows another form of persistent information, a record of *attributes* of the entity, to store the lost information, and to store values calculated from the attributes of actions (Jackson, 1983). The *state vector* of a process is an aggregation of the automata-theoretic *general* state and the entity attribute record.

State vector inspection (SVI) is an operation whereby one process can examine the state vector, in whole or part, of another process. The semantics of SVI require that the states obtained are coherent, and this in turn demands that they be unavailable while the inspected machine is involved in making a transition between general states. As long as state transitions are regarded as occuring atomically and instantaneously, SVIs can be regarded as orthogonal to the event model. At the implementation stage, this orthogonality will no longer be tenable, and steps will need to be taken to ensure that inspected processes are uninspectable between general states, or else that they explicitly maintain a copy of their last coherent state vectors for inspection purposes (Cameron, 1986).

Adding Functions to the Model

Having produced an animated model of reality, further processes are connected to the specification to meet the functional requirements of the system. Function processes are designed using JSP (Jackson, 1975; King & Pardoe, 1985; Storer, 1987). Function processes transform the contents of their input streams into one or more output streams. Hughes (1979) shows the relationship between JSP-designed programs and a class of finite automata called generalised sequential machines (GSMs). Aspects of her account are now related to the present characterisation of the JSD event model.

The structure of a function process is arrived at by recognising *correspondences* between the structures of the input and output streams and producing a composite structure.

A correspondence defines a translation of nodes on an input structure to nodes on an output structure. For any two nodes to correspond, their descendants must correspond also. An input expression `I` and an output expression `O` correspond if the equality `O = output(I)` can be derived using the following rules, where `R` and `Q` are regular expressions (Hughes, 1979):

```
(i)       R ∈ αI ⇒ output(R) ∈ αO ∪{ε}
(ii)      output(RQ) = output(R)output(Q)
(iii)     output(R | Q) = output(R) | output(Q)
(iv)      output(R*) = (output(R))*
(v)       Rε = εR = R
(vi)      R | Q = Q | R
(vii)     R | R = R
(viii)    (R*)* = R*
(ix)      ε* = ε
```

For example consider the two regular expressions `a(b|c)* d` and `x*y`. If they correspond then `output(a(b|c)*d) = x*y`. That this is the case is shown below:

```
output(a(b|c)* d)
=  output(a)output((b|c)*)output(d)                  (ii)
=  output(a)(output(b)|output(c))*output(d)          (iii, iv)
=  ε(x|x)*y                                          (i)
=  x*y                                               •
```

Having produced a composite structure, a program can be produced by applying a modified read-ahead technique with new rules regarding the placement of input and output statements:

- input statements are arranged to read ahead of leaf nodes of the composite structure derived from the input structure;
- output statements are placed on leaf nodes of the composite structure derived from the output structure.

Figure 3.9 shows a structure diagram for the translation of `a(b|c)* d` into `x*y`, together with the trivial network representation of this problem.

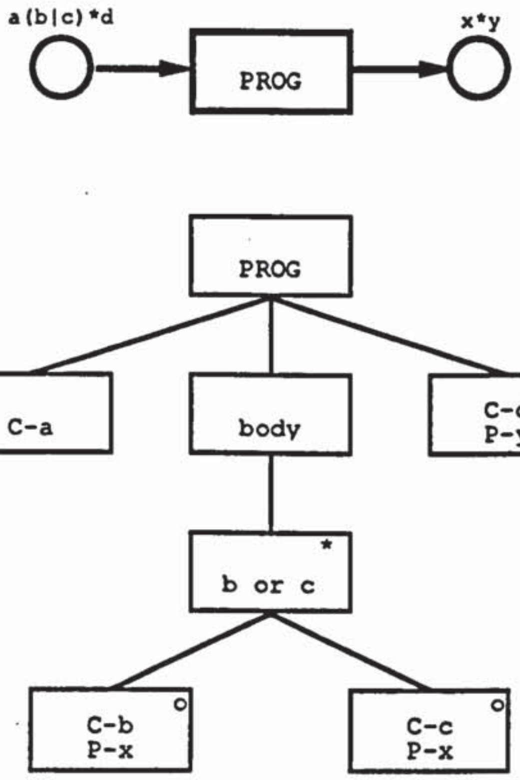


Figure 3.9. A structure diagram representing the translation of a(b|c)* d into x*y, together with relevant network diagram.

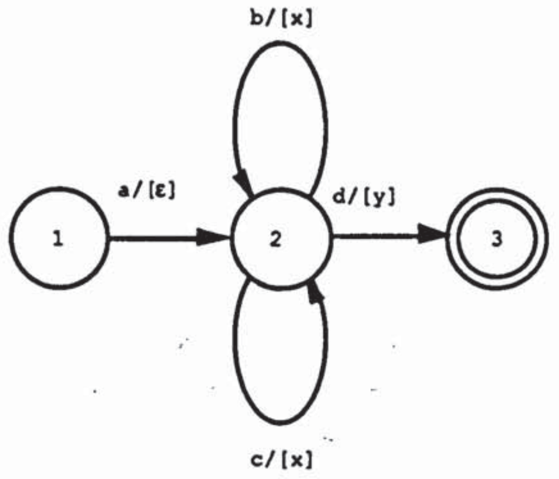
Following Jackson (1975) the consumption or production of records at leaf nodes is denoted by labels of the form `c-"record_name"` or `p-"record_name"` respectively.

Generalised Sequential Machines and JSP

It is now possible to generalise finite automata to allow output. Hughes (1978) employs Ginsburg's (1965) model of a finite automaton with output: the Generalised Sequential Machine (GSM). A GSM is defined by the 6-tuple $(S, \alpha I, \alpha O, \delta, \lambda, q_1)$.

- S is a finite non-empty set of *states*
- αI is an alphabet of inputs
- αO is an alphabet of outputs
- δ is the *transition function* to map $\alpha I \times S \rightarrow S$
- λ is the *output function* to map $\alpha I \times S \rightarrow \alpha O^*$
- q_1 is the start state of S

GSMs are capable of translating strings from an input language, a subset of αI^*, into an output language, a subset of αO^*. Figure 3.10 shows an example of a GSM, together with a transition graph representation, to translate a(b|c)*d into x*y:



```
S = {1,2,3}
αI = {a,b,c,d}
αO = {x,y}
δ = { (1,a,2) (2,b,2) (2,c,2) (2,d,3) }
λ = { (1,a,[ε]) (2,b,[x]) (2,c,[x]) (2,d,[y]) }
q1 = 1
```

Figure 3.10. A GSM to translate a(b|c)*d into x*y

Hughes uses Ginsberg's characterization of GSMs to show that Jackson's basic JSP method is applicable only to GSM computable functions. However, her account of the basic method does not cover all types of function process. She extends her account to cover situations where the correspondence between input and outputs is ambiguous or absent (the so called *structure clashes* (Jackson, 1975)), but leaves those cases where the values of entity or action attributes are used to influence control flow. These situations are discussed in Chapter 5.

3.4 The Implementation Phase of JSD

In principle, JSD specifications are directly executable (Cameron, 1986). This proposition is supported in part by the work of Lewis (1991), Kato & Morisawa (1987) and Warhurst & Flynn (1990) who have built interpreters for parts of the notation, and workers who have shown the correspondence between JSD and CSP (Sridhar & Hoare, 1985; Yeung *et al.*, 1991) for which there exist published formal semantics (Hoare, 1983). Direct execution of JSD specifications is potentially highly inefficient, as such specifications are usually populated by large numbers of sparsely active processes, and system state information tends to be highly distributed. The implementation phase of JSD is concerned with transforming the specification into an efficient form. There is a well-understood repertoire of transformations which are automatable, some of which have been realised in the PRESTIGE workbench (Bass *et al.*, 1991). Later in the thesis, the automata theoretic characterisation is used to develop algorithms for the automation of the *dismemberment* family of transformations, and to support a new transformation technique called *transaction composition*. The rest of this section briefly describes the standard, well-understood JSD implementation issues.

Inversion

If a JSD specification were to be executed directly, with one processor (real or virtual) per process (executing machine), one would observe *dynamic sparsity* (Hull & McKeag, 1984) in the activity of individual processes. Each process would spend most of its time suspended while waiting for input, and processor utilization would be extremely low. The standard transformation in JSD for reducing dynamic sparsity in system execution is *inversion*. Inversion (Jackson, 1983) in its simplest form, transforms two concurrent processes communicating in an asynchronous producer-consumer relationship into a pair of coroutines. Inversion results in implementation routines which express interleaved executions of long-running processes. These 'multiprogrammed' routines are capable of keeping a processor busy even though most of the specification processes from which they are derived are conceptually blocked.

Yeung *et al.* (1991) provide a neat formalisation of inversion in terms of CSP. The description below is essentially as given by Yeung, but for consistency with the rest of this chapter, using the regular expression metalanguage to specify process behaviour rather than CSP-style recursion equations.

Consider the network shown in Figure 3.11.

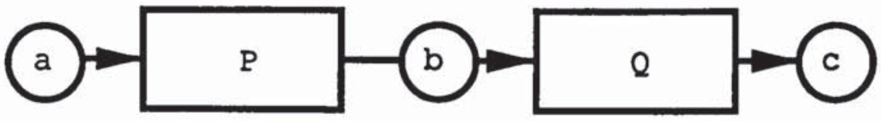


Figure 3.11. A simple network for the purpose of discussing inversion.

Assuming that

```
P = (a?x b!x)*
Q = (b?x c!x)*

where
   a, b, and c are datastreams,
   αa = αb = αc ={1,0},
   the action stream?var denotes input from stream into var,
   the action stream!var denotes output from var to stream,
   actions stream?var & stream!var are synchronised,
```

then direct execution of this system is described by the parallel composition of the two processes, that is P||Q. Rather than implement P and Q as separate concurrent processes, they can be combined into a single sequential thread by arranging for their executions to be interleaved. In this scenario, P runs until it produces a b record, suspends itself and resumes Q to consume the record. Q runs until it next needs a b, suspends itself, resumes P and so on. This pattern of execution can be modelled by adding a common action which synchronises transfer of control from P to Q and back:

```
P' = (a?x b!x ®)*
Q' = (b?x c!x ®)*
```

® must be executed simultaneously by both P and Q , so enforcing coroutine-like behaviour in the system described by P'||Q'. Now if ® together with the i/o actions on channel b are considered as hidden actions, then the new multiprogrammed unit appears to its environment as

```
     (P'||Q')\{b.0,b.1.,®}

=    (a?x c!x)*
```

and can be scheduled as a single process. Figure 3.12 illustrates the JSD notation for the hierarchy of inverted routines obtained in this way.

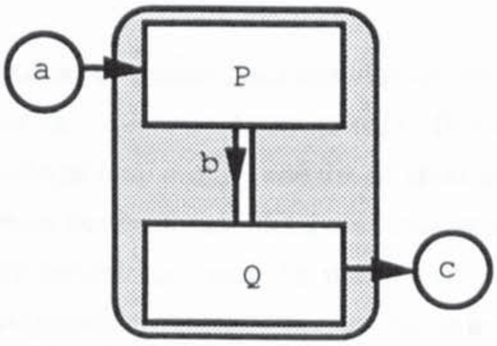


Figure 3.13. Combining processes using inversion.

State Vector Separation

JSD modelling often identifies a *family* of entities (customer orders, aircraft, bank accounts) which correspondingly becomes a set of multiple instances of a process in a specification. At any one time, most, if not all, of these instances will be blocked awaiting communication, and it is therefore highly inefficient to allocate them each to an individual processor. In fact, it is usually necessary to implement the processes as a single re-entrant inverted routine and a separated collection of state-vectors (one for each instance) as a 'database'. The receipt of a message by a process instance then necessitates the retrieval of the appropriate state vector so that the re-entrant routine can be entered in with the correct context. The transformation which accomplishes this rearrangement is called *state-vector separation* and is illustrated together with inversion in Figure 3.12.

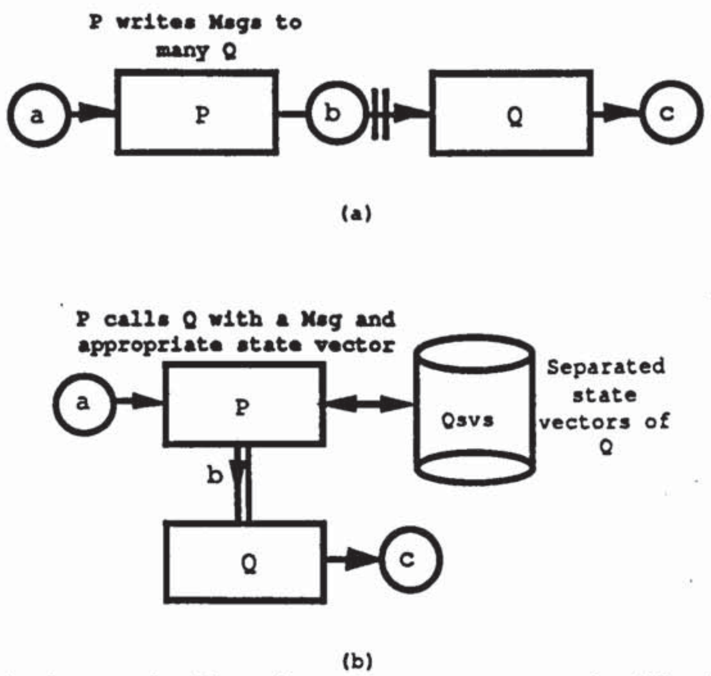

P writes Msgs to many Q
a
P
b
Q
c
(a)
P calls Q with a Msg and appropriate state vector
a
P
Qsvs
Separated state vectors of Q
b
Q
c
(b)

Figure 3.12. A simple example of inversion and state-vector separation (after Bass *et al.*, 1992), showing a simple network (a) before and (b) after transformation.

The requirements of state vector separation place minimal constraints on the implementation of data management in the final system. In some cases (for example, small embedded systems) a state vector database may merely consist of an array of records stored in core memory. In large information systems, such databases may be distributed between central data centres and local processors and may be maintained and updated according to sophisticated network management policies. Further discussion of state vector separation is not pertinent to the rest of the thesis. For a more complete discussion, the interested reader is referred to the standard references (for example, Jackson, 1983; Cameron, 1988).

Dismemberment

The long-running processes obtained at the specification stage often encapsulate functionally unrelated code. In some implementation environments (e.g. transaction processing and concurrent environments) it is valuable, or even necessary, to split up a specified process into a set of modules to be loaded and executed separately. Dismemberment can be used

- to reduce transaction module size when copies of inverted subroutines are used in separate tasks;
- to optimise resource utilization;

- to allow priority scheduling of time-critical portions of a process.

Dismemberment is described in detail in the rest of the thesis, with respect to both its use as an implementation strategy and its automation. A new transformation strategy based on the composition of dismembered components is also developed.

3.5 Discussion

This section addresses two questions raised by our account of the JSD method:

- Given the equivalence of structure diagrams with finite automata, are their situations where one notation is preferable to the other?
- Why restrict the description to be in terms of regular languages?

The Relative Merits of Regular Expressions and DFAs for Specification

A major difference between the regular expression or structure diagram and the finite state machine as representations of behaviour is the kind of thinking they encourage. Regular expressions encourage a static conception of system behaviour, in terms of the possible form of action traces, while finite automata encourage a dynamic view, in terms of transitions between states (Renold, 1988b). Zave & Jackson (1989) discuss the relative appropriateness of the two alternative representations. As has been seen, the states in a finite automaton are abstract and are only implied by its observable behaviour — they cannot generally be associated with anything tangible in the real world. Furthermore, all of the states in a state-transition diagram have equal visual impact, as do the various paths between them. A state-oriented specification can therefore contain many elements which convey little tangible meaning, especially to a non-specialist reader. Actions on the other hand are clearly associated with real world events and are therefore easy to identify and model. When subsequences of actions reflect particular sub-behaviours in the real world (see for example `Loan Part` in the library BOOK example in Figure 3.5) a structure diagram can represent the close association among the actions in the subsequence in a way that a transition diagram cannot. Structure diagrams are therefore to be favoured when a specification is dominated by consideration of action sequences.

On the other hand, there are processes whose behaviour is dominated by a consideration not of action sequences, but of system status. Consider the automaton in Figure 3.14, which is adapted from an example given by Zave and Jackson. The automaton specifies part of a call-forwarding system for a telephone. According to this specification, the call-forwarding facility can be toggled on or off by pressing the 3 key while the receiver is off the hook. An equivalent regular expression for this specification is `(off_hook 3* on_hook)*` which, while clearly specifying the same regular language, conveys no impression of the status of the call forwarding facility. Zave & Jackson's best attempt at producing a structure diagram specification which indicates status employs three trees with a total of 25 nodes (and is judged too large and complicated to include in their paper).

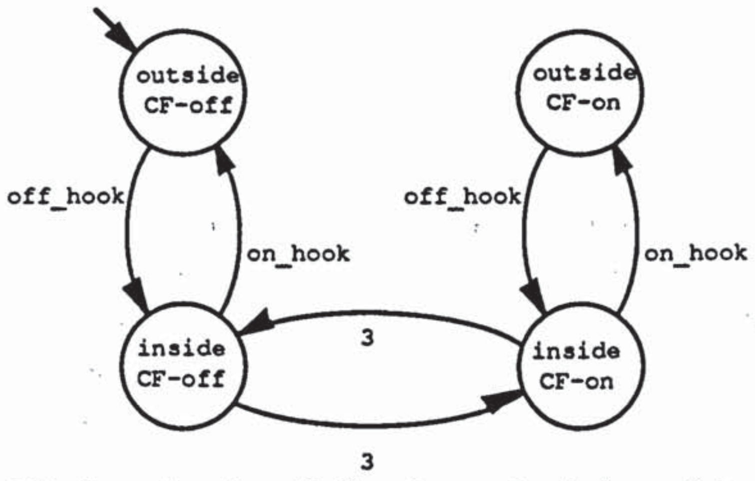


Figure 3.14. State-oriented specification of a part of a telephone call forwarding system, after Zave & Jackson (1989).

Broy (1991) sees actions and states as two sides of the same coin and recommends taking an action-oriented or state-oriented viewpoint as appropriate to the problem. Zave & Jackson (1989) illustrate a method of combining such views in single specifications.

More Powerful Grammars

Consider the grammar introduced earlier with the production rules

$$
\begin{aligned}
S &\rightarrow (\; R \\
S &\rightarrow \varepsilon \\
R &\rightarrow S \;) \; .
\end{aligned}
$$

This grammar is a *context-free* (or type 2) grammar representing arbitrarily deep nests of balanced parentheses. The level of recursion keeps an implicit count of the number of unclosed parentheses and 'returning' from each application of R ensures the generation of the appropriate number of closing parentheses. As the definition of the grammar allows an arbitrary number of applications of R , in arbitrary nested combinations, it is impossible to construct a finite state graph for S. It is however possible to construct a recursive transition graph such as the one in Figure 3.15.

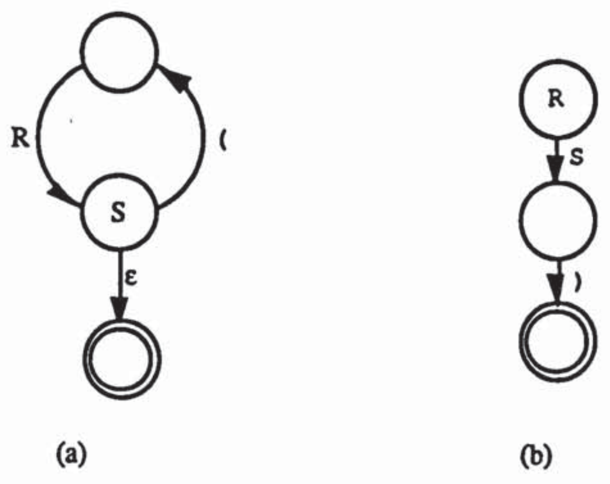


Figure 3.15. A recursive transition network.

The subgraph marked (a) represents a top level recogniser for the language S. It will accept either an ε or an opening parenthesis followed by an instance of R. To accept an R, the subgraph marked (b) must be entered somewhat in the manner of a subroutine. As R is entered immediately after each opening parenthesis, and accepts (as its last action) only a single closing parenthesis, balancing is guaranteed. Minsky (1967) describes a class of automata called the pushdown automata, which recognise context-free grammars in a similar way.

A structure diagram for S is shown in Figure 3.16. Recursion is represented by naming a leaf node with the name of a non-terminal node from elsewhere in the tree (Cameron, 1988).

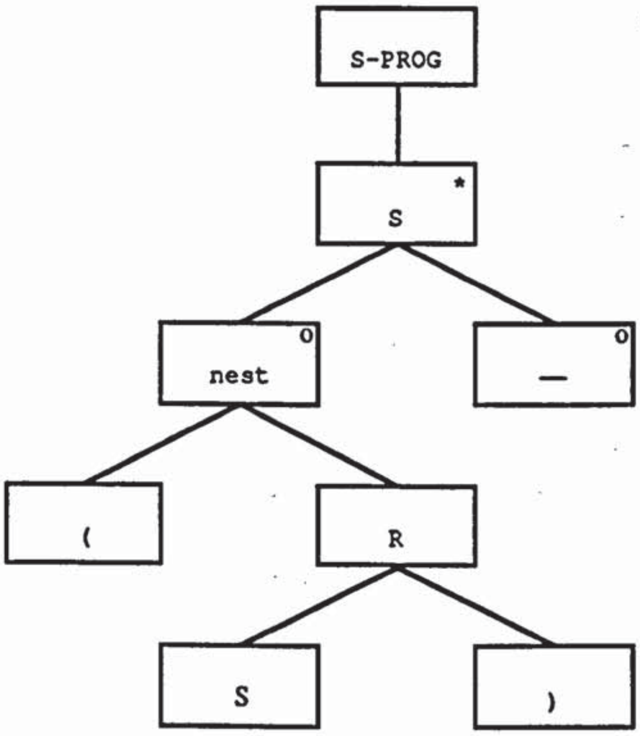


Figure 3.16. A structure diagram with recursion for accepting a context-free language.

The code which would result from applying the read ahead rule to this tree is as follows:

```
S-PROG itr while not end of stream
read(input);
   S sel input = "("
      read(input);       -- accept "("
      R seq
         S;
         read(input);    -- accept ")"
      R end;
   S alt
      skip;              -- accept "-"
   S end;
S-PROG end.
```

Dwyer (1991), one of those involved in the early development of JSP, has commented on the close relationship of JSP to LL(1) parsing by recursive descent (Bornat, 1979). That the main sources on JSD make no reference to this correspondence can be taken as evidence that the extra expressiveness afforded by context free structure diagrams is not worth the added complication (certainly also, common JSD target environments such as early COBOLs and many embedded systems mitigate against the use of recursion).

JSD provides the power lost by its reliance on regular structures by introducing further parallelism. One way to look at the parentheses example is in terms of the wish to open new levels of nesting before closing old ones. In other words, it is required to have several processes with the structure shown in Figure 3.17 active at any one time. The solution is to view each pair of parentheses in a nest as a so-called *marsupial* process (Jackson, 1983) running concurrently with each other pair. This solution allows arbitrary nesting while guaranteeing balancing.

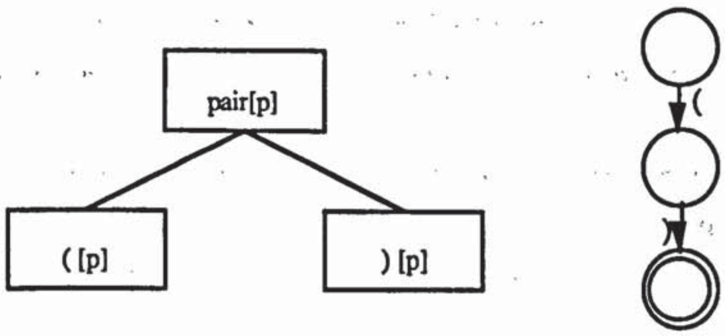


Figure 3.17. Marsupial process for the nested parentheses problem.

s is then represented as a synchronising process which accepts a sequence of opening or closing parentheses as shown in Figure 3.18. Each such parenthesis is identified with a particular nest by a unique identifier. Balancing of each parenthesis pair is guaranteed by the structure of the corresponding `pair[n]` process. This approach arbitrarily increases the number of processes in the specification, but presents no more conceptual difficulty than understanding the indeterminate number of nested procedure calls implied by a recursive specification. In less contrived examples, it is a useful way of discovering entities which were overlooked in the initial stages of the development (Jackson (1983) provides various examples).

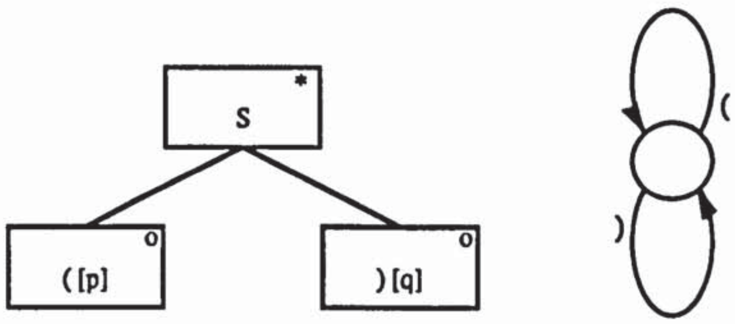


Figure 3.18. Synchronising process for the nested parenthesis problem.

3.6 Conclusion

JSD has been characterised in terms of an event model based on the theories of formal languages and finite automata. This description of JSD forms the basis of the contribution of the thesis — the application of results and techniques from well-established subdomains of computer science to the contemporary problem of implementing desirable, currently unsupported (yet sometimes manually performed) transformations on JSD specifications.

Parallels have been drawn between regular languages and entity life histories, and regular grammars and expressions have been used to model entity behaviour. Finite automata have been introduced as a way of viewing the animation of such models. The modelling of individual entities had been related to the Network and Implementation phases of JSD, drawing on work by Hughes (1979) on the formalisation of JSP, and Sridhar & Hoare (1985) and Yeung *et al.* (1991) on the relationship between JSD and CSP. The chapter concluded with a comparison of structure diagrams and finite automata as modelling notations, and considered the use of context-free grammar as a means of gaining extra expressiveness. It was shown how JSD uses concurrency as a way to gain the extra power afforded by the more sophisticated class of grammar while staying with the simplicity of regular structures.

Chapter 4
Transforming Structure Diagrams into Automata

4.1 Introduction

The previous chapter related JSD processes and finite automata. Lewis (1991) describes an approach to JSD process implementation in object-oriented environments which depends on the implicit construction of a restricted class of nondeterministic finite automaton. This chapter makes the automata theory underlying Lewis's approach explicit and is therefore able to explain its limitations. The subsequent discussion draws on research which has been carried out in the area of automatic construction and manipulation of automata to develop an improved approach which, while based on Lewis's original insight, addresses problems he did not cover.

4.2 Followsets

Two equivalent views of JSD processes have been presented and related: a tree notation for regular expressions, and finite automata. Both have been shown to constrain the syntactic structure of strings. They do this by restricting the possibilities for each symbol in a string based on the form of the symbol's prefix in the string. For example, consider the regular language

$$L = a\ (b\ c\ |\ d\ e) \qquad \alpha L = \{a,b,c,d,e\}\ .$$

The following predicates about strings $S,\ T \in L$ are true:

$$\langle a,\ b\rangle \leq S \Rightarrow S(3) = c$$

$$\langle a\rangle \leq T \Rightarrow T(2) \in \{b,d\}\ .$$

As a second example, consider the language

$$M = w\ x\ y^*\ z \qquad \alpha M = \{w,x,y,z\}\ .$$

Again, it is possible to write predicates relating prefixes to subsequent symbols in strings such as $U,\ V \in M$. For example,

$$\langle w,x\rangle \leq U \Rightarrow U(3) \in \{y,z\}$$

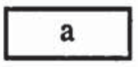
$$\langle w,x,y,y,y\rangle \leq V \Rightarrow V(6) \in \{y,z\} \;.$$

In general for any language L and any occurrence of a symbol $s \in \alpha L$ there will be a set of symbols $F \; (\alpha L \supseteq F)$, which can immediately follow it, determined by the grammatical structure of L. F is said to be the *follow set* of s. Jackson (1983) is the first source to discuss follow sets in connection with JSD, though their use is well established in the field of parsing theory (see for example Aho *et al.*, 1985).

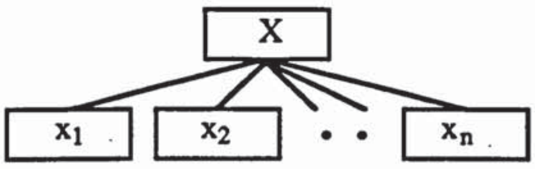
Figure 4.1 gives a partially graphical presentation of rules based on those given by Jackson (1983), defining two functions First and Follow,

```
First  : StructureDiagram →  setof Actions
Follow : StructureDiagram →  setof Actions   .
```

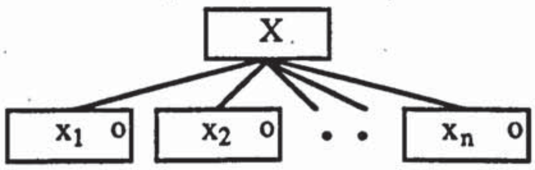
These functions compute the *first sets* and follow sets of any Jackson (sub)tree. The first set of a tree contains all those actions which can begin a string in the (sub)language the tree represents, while the follow set contains all those actions which can immediately follow the same set of strings. Jackson leaves the follow set of the root of a tree undefined. Lewis (1991) adds another rule (Follow(root) = {}) to deal with this case, while Jackson requires that an *end of file* marker be sequentially composed with the tree (and by implication adds the rule Follow(eof) = {}). This thesis follows Jackson as it simplifies the algorithm for DFA construction presented later (see Section 4.7).



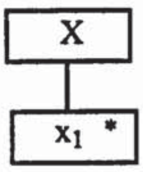
a is a leaf node
First(a) = {a}



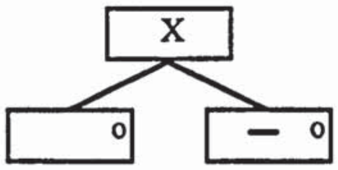
X is a sequence node
First(X) = First(x_1)
Follow(x_n) = Follow(X)
Follow(x_i) = First(x_{i+1})
for $1 \leq i < n$



X is a selection node
First(X) =
First(x_1) ∪ First(x_2) ∪..∪ First(x_n)
Follow(x_i) = Follow(X)
for $1 \leq i \leq n$



X is an iteration node
First(X) = Follow(x_1)
= First(x_1) ∪ Follow(X)



— is a null leaf node
First(—) = Follow(—)

Figure 4.1. Definition of `First` and `Follow`.

Figure 4.2 illustrates the effect of applying `First` and `Follow` to each of the subtrees of the ACCOUNT structure. Each node is labelled with the first and follow sets of the subtree of which it is the root (follow sets are in bold).

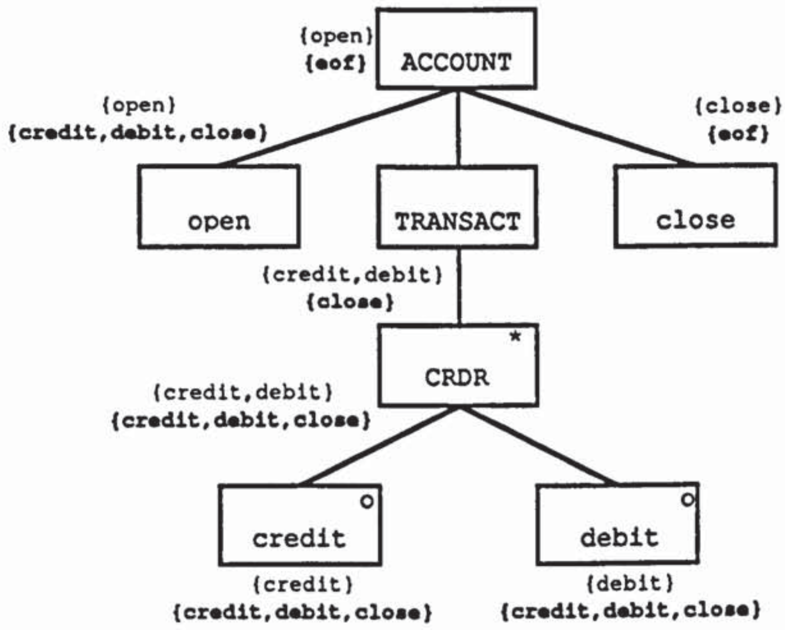


Figure 4.2. Application of `First` and `Follow` to the ACCOUNT structure.

It is possible to enumerate a mapping from leaf nodes of a structure diagram to follow sets. For example, the *follow map* of the ACCOUNT process is

```
{(open   → {credit, debit, close}),
 (credit → {credit, debit, close}),
 (debit  → {credit, debit, close}),
 (close  → {eof})}  .
```

Lewis (1991) describes an algorithm to construct the follow map of a process. He uses this representation as the basis for the execution of JSD processes in a Smalltalk-80 environment. The following algorithm shows how a follow map can be used to drive a regular language recogniser and so illustrates the idea behind Lewis's approach:

```
procedure recognise(S : StructureDiagram):boolean;
begin
   contextset := First(S);
   followmap := followmap(S);
   read(insym);
   while contextset <> {eof} do
      if insym ∈ contextset then
         contextset := followmap(insym);
         read(insym)
      else
         return false
      endif
   endwhile
   return true
```

```
end recognise.
```

There is a parallel between the idea of a followmap and that of *successor functions* in the PAISLey notation (Zave, 1982) which was discussed in Chapter 2.

Lewis observes that a particular process can be implemented in a *hard-coded* form which incorporates the followmap information into the guards for an iterated multibranch conditional. A control variable keeps track of the current context. Below is such a text for the ACCOUNT process:

```
procedure follow_mapped_account;
begin
   state := open;
   read(input);
   if state = open and input = "credit" then
        read(input);
        state := credit
   elseif state = open and input = "debit" then
        read(input);
        state := debit
   elseif state = open and input = "close" then
      read(input);
      state := close
   elseif state = credit and input = "debit" then
      read(input);
      state := debit
   elseif state = credit and input = "close" then
      read(input)
      state := close
   elseif state = credit and input = "credit" then
      read(input);
      state := credit
   elseif state = debit and input = "close" then
      read(input);
      state := close
   elseif state = debit and input = "credit" then
      read(input);
      state := credit
   elseif state = debit and input = "debit" then
      read(input);
      state := debit
   elseif state = close and input = "eof" then
      skip;
   endif;
end follow_mapped_account.
```

The `recognise` algorithm for interpreting followmaps given above can be modified to generate such hard-coded implementations of processes. In outline, such an algorithm would have the following form (where the carat symbol '^' is used to obtain the value of a variable to be substituted into a string):

```
procedure hardcode(S : StructureDiagram )
begin
   followmap := followmap(S);
   emit( 'read(insym);' );
   emit( 'while state <> eof do'    );
   for each maplet (sym → fs) in followmap do
```

```
      for each s in fs do
         emit( 'if state = ^sym and input = ^s
                  then state := ^s;
                  read(insym);
               endif;');
      endfor
   endfor
   emit( 'endwhile;' )
end hardcode.
```

Systematic Error Checking Based on Followmaps

Jackson (1983) shows how knowledge of the context set of each node can form the basis of a systematic error recovery scheme. Model processes represent only those possible life-histories which the developer has abstracted from the real world. They therefore expect their input to conform to one of these life-histories. If, for any reason, action messages arrive in an order other than the one prescribed by the process structure (perhaps an operator has entered a message incorrectly, or a sensing device is malfunctioning) then the behaviour of the process is undefined. To avoid this situation occurring in an implemented system, *context filter* processes are positioned upstream of the model processes in 1-1 correspondence. Each context filter checks the current context set of its model process and passes on acceptable messages only. If a context error occurs, the filter issues a diagnostic report and begins to skip messages until it finds one in the context set. The method is similar to the context-based symbol skipping error recovery schemes found in syntax directed program translators (Aho *et al.*, 1985; Bornat, 1979). Below is the text of a context filter for the ACCOUNT process (symbol skipping loops are shown in italics):

```
procedure account_filter;
begin
   read(msg);
   while msg ∉ {open} do read(msg) endwhile;
   write(msg); -- open
   read(input);
   while msg ∉ {credit, debit, close} do read(msg) endwhile;
   while (msg = credit) or (msg = debit) do
      if msg = credit then
         write(msg); -- credit
         read(msg);
         while msg ∉  {credit,  debit,  close}  do  read(msg)
endwhile;
      elsif msg = debit then
         write(msg); -- debit
         read(msg);
         while msg ∉  {credit,  debit,  close}  do  read(msg)
endwhile
      endif
   endwhile
   write(msg) -- close
end account_filter;
```

Context filters are discussed further in Chapter 6.

4.3 Limitations of Direct Followmap Interpretation

This section discusses the limitations of what will, from now on, be termed *followmap interpretation* (FMI). Subsequent sections show how by drawing on the language theory underlying this approach, it is possible to substantially address these limitations. Lewis himself identifies three problems with FMI:

- no distinction is made among multiple occurrences of actions at the leaves of a tree;
- implementations can be very inefficient;
- *recognition difficulties* (also known as *backtracking* problems (Jackson, 1975)) arising from a choice of transitions for a given action are not addressed.

The first two of these issues are expanded on below. Treatment of recognition difficulties is deferred until Chapter 5 as it depends on further theoretical prerequisites presented later in this chapter.

No Distinction Among Multiple Occurrences of Actions

Although it is uncommon in the standard examples of (model) structure diagrams given in the literature, it is perfectly permissible to label more than one leaf with the same action name. Of course, in lexical analysis this is the rule rather than the exception (for example, the set of all Pascal identifiers might be described as `letter(letter|digit)*`), and indeed it is very common in JSD *function* processes (and JSP programs). In cases where there are multiple occurrences of action names, a followmap which maps names to sets of names will not provide a correct basis for an implementation. Consider the regular expression `(a|b)*cab`. Each terminal letter can be associated with a set of positions at which it can legally occur (Aho *et al.*, 1985). For example, the letter `a` can appear at both position `1` and position `4`. Clearly, the follow set of the `a` at position `1` is different to that of the `a` at position `4`. If this distinction is not allowed for, then the follow set of `a` in the followmap will depend on the order of evaluation of the follow sets of the leaves. This will lead to an incomplete representation of the process. The problem is easily rectified by computing the function `followpos` (`position` $\rightarrow$ **`setof`** `positions`) and a naming mapping (`position` $\rightarrow$ `name`) which binds action names to positions. The followpos function and attendant naming mapping for `(a|b)*cab` are

```
followpos((a|b)*cab)= { 1 → {1, 2, 3},
                        2 → {1, 2, 3},
                        3 → {4},
                        4 → {5},
                        5 → {6}      }.

namemap((a|b)*cab))= {  1 → a,
                        2 → b,
                        3 → c,
                        4 → a,
                        5 → b,
                        6 → eof }    .
```

The extra information available in this form can be used to support the correct recognition of languages with multiple synonymous terminals by modifying the program `recognise` given earlier:

```
procedure  recognise2(S  :  StructureDiagram):  boolean;
begin
   contextset := FirstPos(S);
   contextsyms := symbols(contextset);
   read(insym);
   while contextsyms <> {eof} do
      if insym ∈ contextsyms then
         contextset := followpos(incontextpositionof(insym));
         contextsyms := symbols(contextset);
         read(insym)
      else
         return false
      endif
   endwhile
   return true
end recognise2.
```

A similar modification can clearly be made to the `hardcode` algorithm given earlier.

Inefficient Implementations

The main cause of inefficiency in a direct followmap implementation such as the one shown earlier for ACCOUNT, is the suboptimal number of possible values of the `state` variable. Code size is larger than necessary, as there must be (often duplicated) input and state update statements for each of these values. There is a corresponding requirement to evaluate more branch conditions than theoretically necessary to decide which transition to make. Lewis (1991) suggests two optimisations: factoring out evaluation of state into a case statement to allow a table-based evaluation of that component of the branch conditions, and ordering branches so that more likely conditions are evaluated first (assuming a suitable ordering heuristic can be found). The first of these can easily be accommodated in the `hardcode` algorithm. All that needs to be done is to move the generation of statements of the form `if state = x` outside the innermost loop, and change them into `case` syntax:

```
procedure hardcode2(S : StructureDiagram)
begin
   followmap := followmap(S);
   emit( 'read(insym);' );
   emit( 'while state <> eof do'    );
   emit( 'case state of' );
   for each maplet sym → fs in followmap do
      emit( '^sym => ');
      for each s in fs do
         emit( 'if imput = ^s
                  then state := ^s;
                  read(insym);
                  endif;')
      endfor
   endfor
   emit(       'endcase
            endwhile;'  )
end hardcode2.
```

The effects of these measures are likely to be small, however, especially when judged against the gains which can be achieved through optimising the range of values of `state`, i.e., through state-optimising the finite automaton underlying the followmap representation.

The reason Lewis's approach produces an automaton with a suboptimal number of states is because it introduces a unique state to denote the context the machine is in *after each leafnode* has matched an action. So, the number of states will always equal `|firstpos(root)| + number of leaves`, regardless of whether or not this number of states is the minimum possible for an automaton recognising the same language. Poo (1991) and Cameron (1988) use the same state-introduction rule as Lewis, suggesting that the states be named systematically by past tense form of the name of the action, so that `loan ⇒ loaned, arrive ⇒ arrived`, and so on (note, incidentally, how this naming convention perpetuates the synonymous leaf node problem, as it creates states corresponding to distinct positions and names them with the same identifier) . Figure 4.3 shows a structure diagram for a library BOOK structure (Cameron, 1986). The vertical arrows label the states introduced by following Poo's method, and implied by an FMI approach.

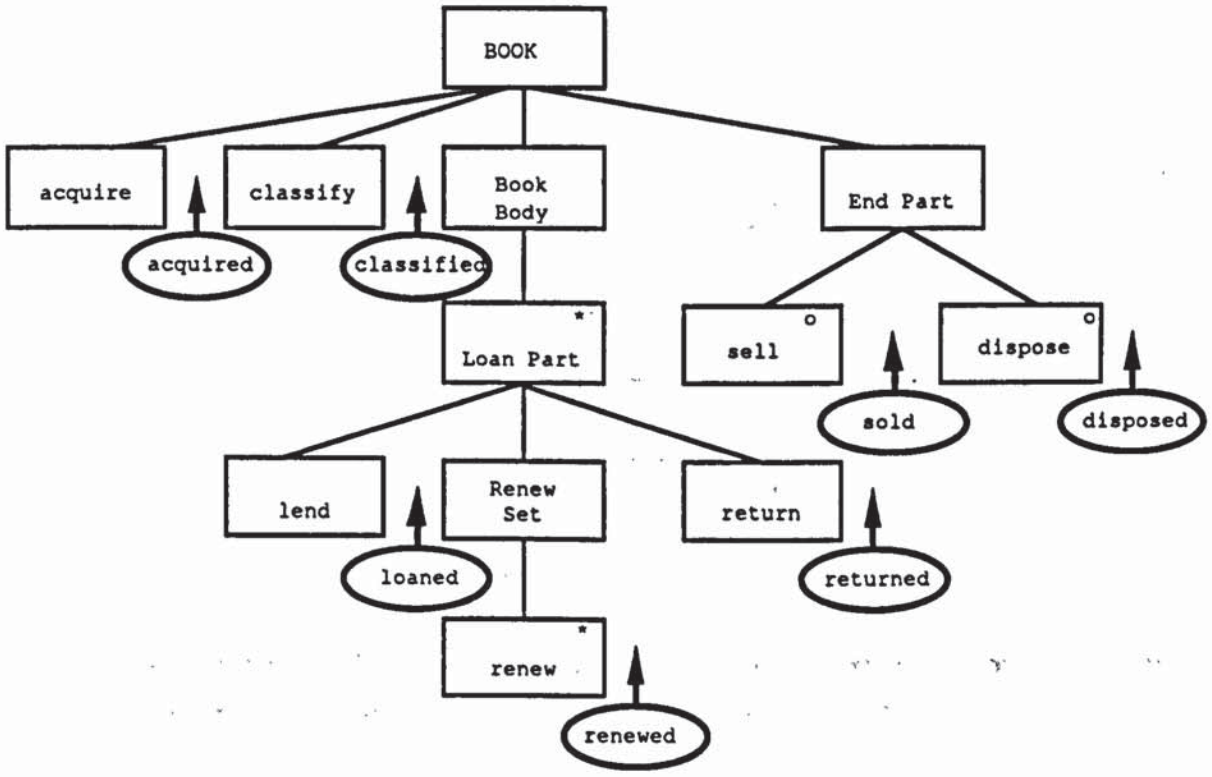


Figure 4.3. The library book structure together with implied states following Poo (1991).

Note that some of the states marked in figure 4.3 are equivalent in the sense used in Chapter 3 to define the notion of state (that is, they cannot be distinguished by subsequent input). For example, `classified` ≡ `returned` and `sold` ≡ `disposed`. In general, the states following the last nodes of a selection are equivalent, and the states immediately preceding or finishing an iterated part are equivalent. Indeed, any two leaf nodes whose `followpos` values are equal must both represent transitions into the same state (this is intuitively the case, but later we present a more rigourous argument and use it to underpin the explanation of a systematic approach to state minimisation). Figures 4.4 and 4.5 illustrate this point for selections and iterations respectively.

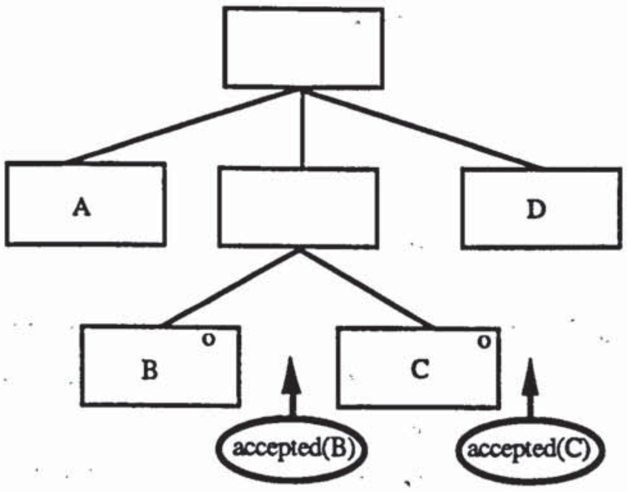


Figure 4.4. Equivalent states after a selection.

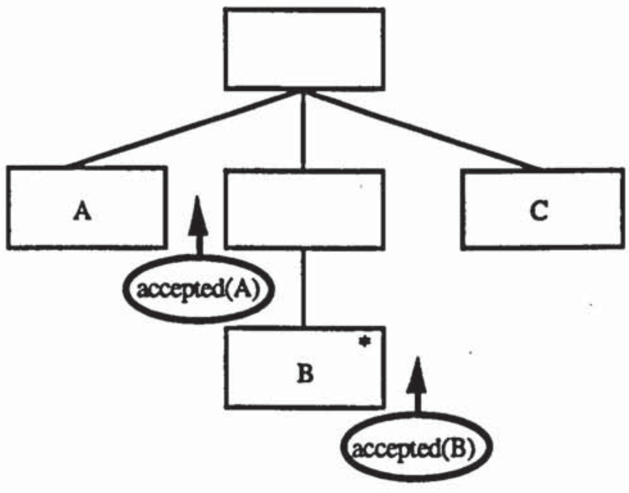


Figure 4.5. Equivalent states after an iteration.

Consider once again, the bank ACCOUNT example. Figure 4.6 shows its transition graph produced from its followmap and labelled in the manner of Poo (1991)

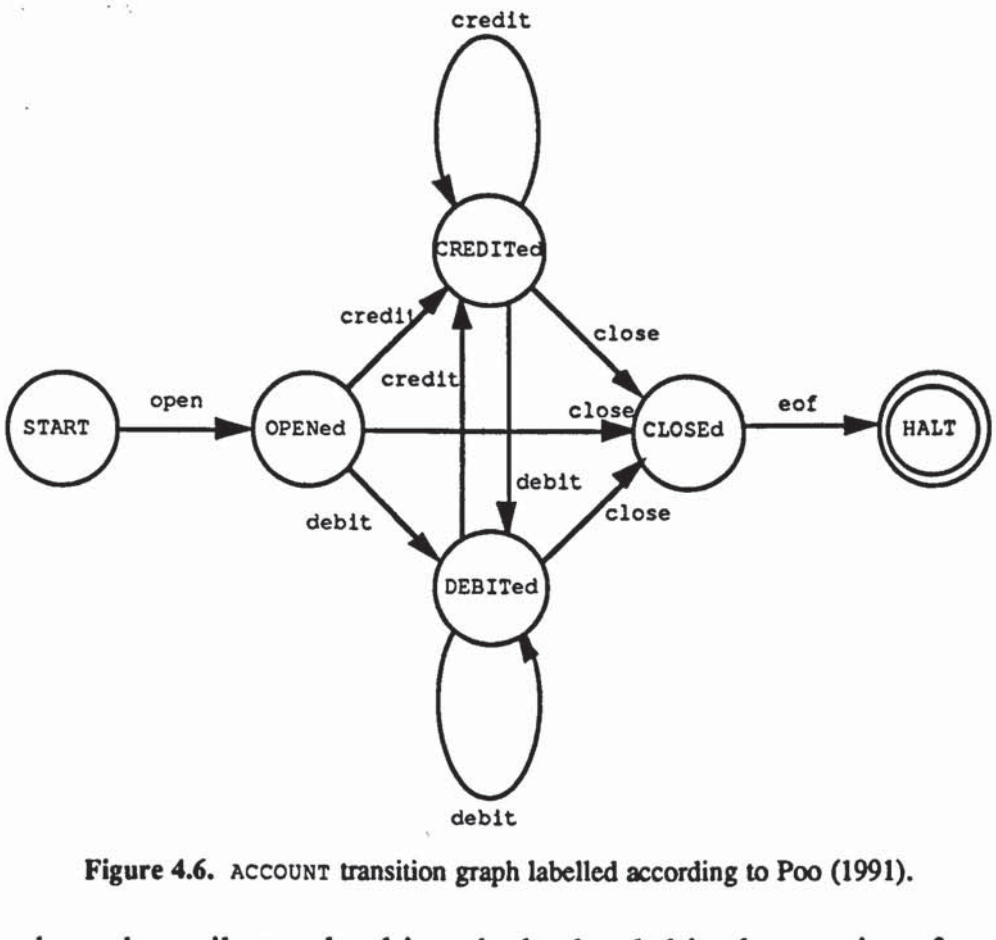


Figure 4.6. ACCOUNT transition graph labelled according to Poo (1991).

This graph can be easily translated into the hard-coded implementation of ACCOUNT, namely `follow_mapped_account`, given earlier.

Clearly, `opened` ≡ `credited` ≡ `debited`, as ACCOUNT behaves identically for a given string regardless of which of the three states it starts from. Recognising this, the ACCOUNT transition graph presented in Figure 4.6 above can therefore be simplified to that shown in Figure 4.7.

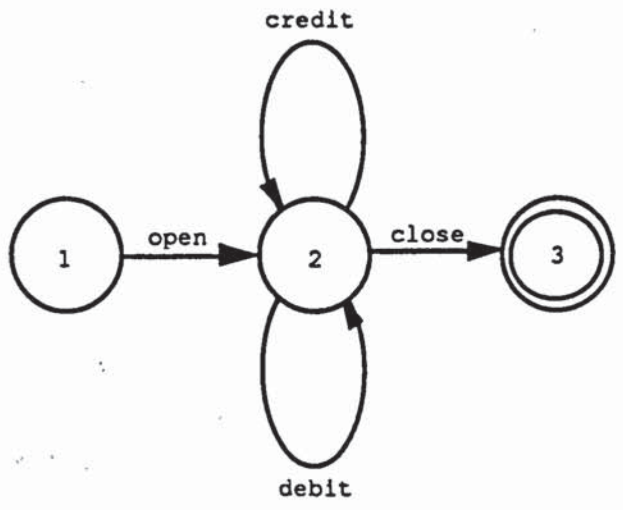


Figure 4.7. Minimum state ACCOUNT transition graph.

This graph gives rise to the following efficient hard-coded implementation:

```
procedure min_state_account;
begin;
   state := 1;
   read(input);
   while input <> eof
      case state of
         1 =>
            case input of
               open =>
                  read(input);
                  state := 2
            endcase;
         2 =>
            case input of
               credit =>
                  read(input);
                  state := 2
               debit =>
                  read(input);
                  state := 2
               close =>
                  read(input);
                  state := 3
            endcase;
         3 =>
            case input of
                  eof =>
                     skip;
            endcase
      endcase
   endwhile
end min_state_account.
```

In rest of the chapter the construction of an explicit NFA from a `followpos` representation is discussed. This is followed by a discussion of an algorithm for building an equivalent DFA from any NFA. Next, the issue of state minimisation in DFAs is introduced. Finally these three strands are brought together in the form of an algorithm for the direct construction of a DFA from a structure diagram which also performs some state minimisation.

4.4 Conversion of a Followmap into an NFA

Below is tabulated the *followpos* function for ACCOUNT (where ACCOUNT is sequentially composed with a final `eof` node), obtained by producing the follow map of its structure diagram where the leaves have been renamed with their inorder positions:

```
{(1   → {2, 3, 4}),
 (2   → {2, 3, 4}),
 (3   → {2, 3, 4}),
 (4   → {5})}  .
```

This table, together with the renaming mapping

```
nameof: position → name
   {(1 →  open)
    (2 →  credit)
    (3 →  debit)
    (4 →  close)
    (5 →  eof)}  ,
```

can be used to produce a ε-transition-free NFA, provided that, following Aho *et al.* (1986),

- all positions in `First(root)` are considered start states;
- for each pair `(i,j)` such that `(followpos(i) = S)` $\wedge$ `j` $\in$ `S`, an edge is added linking state `i` to state `j`;
- each edge `(i,j)` is labelled by `nameof(j)`;
- the position associated with `eof` is the only accepting state.

Consider now another example, the regular expression `(a|b)*ac`. Figure 4.8 shows the automaton produced by applying the above rules.

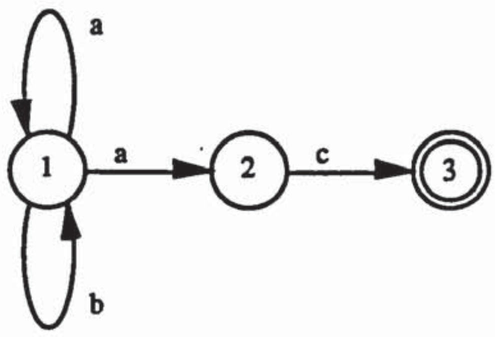


Figure 4.8. An NFA recognising `(a|b)*ac`.

Note that this automaton is nondeterministic. That the FMI approach produces nondeterministic automata is hidden in Lewis's (1991) thesis, as the specific examples he presents all give rise to deterministic machines (the set of DFAs is a subset of the set of NFAs). Were Lewis's algorithm to be applied to `(a|b)*ab`, the resulting program would be incorrect. Consider the pertinent fragment

```
if state = 1 and input = a then
   state := 1;
elsif state = 1 and input = a then
   state := 2
```

An important theoretical result in automata theory is that for any NFA, there is a corresponding DFA which recognises the same language (see for example Rayward-Smith, 1983). Construction of a DFA forms the basis of a solution to the problem of catering for recognition difficulties. This topic is dealt with in detail in Chapter 5.

4.5 Conversion of NFAs to DFAs

This section describes the conversion of NFAs into corresponding DFAs by *subset construction* (Aho *et al.*, 1986) and lays the framework for discussion of an algorithm which constructs a size-optimised DFA directly from a structure diagram. The basic idea behind the subset construction is that for an NFA `N`, a corresponding DFA `D` is constructed such that each deterministic state of `D` , or *`Dstate`*, represents the subset of the non-deterministic states of `N` (*`Nstates`*) that `N` could be in after a particular sequence of inputs `s`. In effect, `D` tracks all possible paths through `N` on `s` *in parallel.* Figure 4.9 shows a very simple NFA (a) and its corresponding DFA (b).

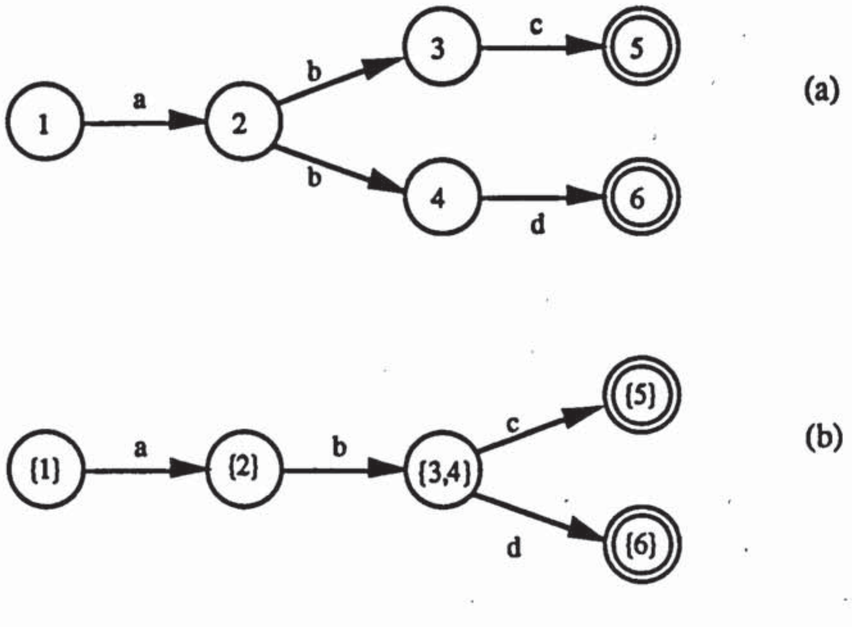
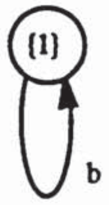


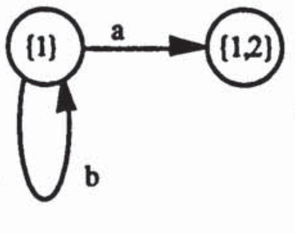
Figure 4.9. An NFA (a) and an equivalent DFA (b).

In the following discussions, names of the form $\texttt{Dstate}_x$ and $\texttt{Nstate}_y$ denote `Dstates` or `Nstates`, labelled by `x` or `y` respectively, in the associated diagrams. In this simple example, $\texttt{Dstate}_{\{3,4\}}$ represents the subset of `Nstates` ($\{\texttt{Nstate}_3, \texttt{Nstate}_4\}$) that the NFA could be in having received input from the language `ab`.

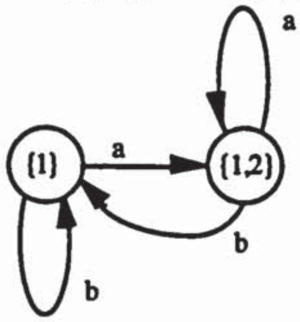
Consider now the example shown above in Figure 4.8. This NFA — call it `N` — recognises the language `(a|b)*ac`. An informal description of the way the subset construction builds an equivalent DFA `D` is now presented. If `D` is to be deterministic, each of its states must have at most one out-transition on each input symbol. In $\texttt{Nstate}_1$ there is no transition on `c`, and only one on `b`, so we can start to construct `D` as follows:



In $\texttt{Nstate}_1$ the only potential nondeterminism arises from the input symbol `a`. The input symbol `a` can cause a transition from $\texttt{Nstate}_1$ to either $\texttt{Nstate}_1$ or $\texttt{Nstate}_2$. This subset of the `Nstates` is represented by the single new $\texttt{Dstate}_{\{1,2\}}$:



This completes consideration of $\texttt{Dstate}_{\{1\}}$. Next the algorithm considers $\texttt{Dstate}_{\{1,2\}}$, firstly as if it were $\texttt{Nstate}_1$ and then as if it were $\texttt{Nstate}_2$. The transitions from $\texttt{Nstate}_1$ have already been considered (a → {1,2}, b → {1}). D can be elaborated accordingly:



(Note that at this stage, the subset construction algorithm has eliminated the nondeterminism in the partition of N recognising (a|b)*, and constructed a deterministic version.) The transitions for $\texttt{Nstate}_2$ are very simple (c → {3}) and give rise to no nondeterminism, so D is completed by the addition of a transition from $\texttt{DState}_{\{1,2\}}$ to $\texttt{DState}_{\{3\}}$ on input c. Figure 4.10 shows the transition graph for the complete version of D.

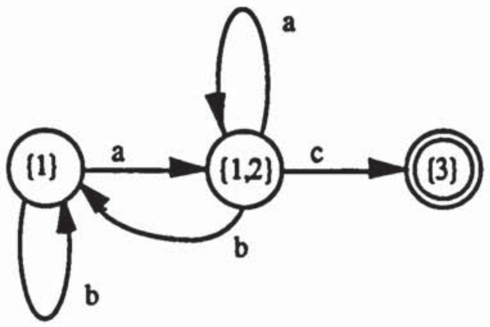


Figure 4.10. A DFA accepting the same language as the NFA in Figure 4.8.

The discussion so far has considered only nondeterminism arising from multiple transitions on the same symbol. The approach can be extended in a straightforward manner to include NFAs with ε-transitions. The *ε-closure* of a state is the set of states reachable on ε-transitions alone. When constructing the subset of Nstates to which a particular input symbol s can lead, the ε-closure of these states must be included. Further consideration of ε-transitions and ε-closures is delayed until the next chapter. For the present examples, none of which contain ε-transitions,

```
ε-closure(δ(state, input)) = δ(state, input).
```

Below is the subset construction algorithm basically as given by Aho *et al.* (1986).

```
procedure subset_nfa;
   Dstates := { ε-closure(s0) };
   while there is an unmarked state T ∈ Dstates do
      mark(T);
      for each input ∈ αM do
```

```
            U := ε-closure( δ(T, input) );
            if U ∉ Dstates then
               Dstates := Dstates ∪ { U };
            endif;
            Dtran[T, input] := U
        endfor
    endwhile
end subset_nfa.
```

Note that in the worst case, the number of `Dstates` is exponential in the number of `Nstates` as there are 2^k subsets of states for a k-state machine. In JSD applications, this is extremely unlikely to arise. Large numbers of subsets only occur when converting machines with high non-deterministic branching factors. In the case of JSD structures, this can occur only when (improbably many) multiple recognition difficulties are expressed in a single structure. Recognition difficulties and backtracking are discussed in the next chapter.

In JSD applications, happily, subset construction can *reduce* the number of states in a DFA relative to the NFA implied by a followmap, by allowing subsets to be identified as representing equivalent states. This is discussed further as the chapter proceeds.

4.6 State Minimisation

For any finite automaton, there exists a minimum-state DFA which recognises the same language (Backhouse, 1979). For any DFA $M = (S, \alpha M, \delta, F, q_1)$, a string s in αM^* is said to *distinguish* a pair of states $p, q \in S$, if there exists a path from p to a state in F on input s, but no such path from q, or *vice versa*. The simplest example is provided by the empty string ε, which distinguishes any accepting state from any non-accepting state. Note that this notion of distinction defines a partition of the states of a machine. In the case of ε above, the two groups of this partition are F and $S \backslash F$.

Any two states of a machine which are indistinguishable by any input can be considered equivalent — recall from Chapter 3 that a state is precisely an equivalence class of histories which cannot be distinguished by any string. This insight provides the basis for a state minimisation algorithm (Aho *et al.*, 1985). The algorithm starts from the 'accepting end' of the machine and works backwards from the initial partition created by ε, taking each group in turn and attempting to distinguish the elements of that group with one of the possible input symbols, so progressively refining the partition until none of the groups of states can be further distinguished.

Prior to the partition refinement process, it is necessary to ensure that the transition function δ is *complete*, that is to say that it has a value for all possible combinations of state and input. This can be done straightforwardly by adding a 'dead' state with reflexive transitions for all inputs in αM, and adding a transition to the dead state for all undefined values of δ (Rayward-Smith, 1983). Table 4.1 shows the transition table obtained by applying this procedure to the ACCOUNT process.

Refinement of the partition can now proceed as follows. Call the initial partition Π_1. The algorithm attempts to further split Π_1 into a new partition Π_2, such that for each input symbol, a state in a group G of Π_2 has transitions leading to the same group in Π_1 as any other member of G. That is, all members of G go to the same group of Π_1 on any given input symbol (intuitively, any two states for which this condition does not hold must be capable of being distinguished by a string starting with the symbol which takes them into different groups of Π_1). Once Π_2 has been determined, it becomes considered for further refinement , and the process is repeated until no further refinement is possible. The final partition Π_{final} can now be used to construct a minimum state DFA by picking one state from each group as a representative, discarding the dead state (transitions to the dead state become undefined) and discarding any states not reachable from the start state. A proof that this algorithm produces a minimum state DFA can be found in Hopcroft & Ullman (1979). We now trace its effects on the NFA produced by applying the direct followmap construction to the ACCOUNT process, and show how it leads to the minimized machine shown in Figure 4.7.

	open	credit	debit	close	eof
firstpos(root)	1	dead	dead	dead	dead
1	dead	2	3	4	dead
2	dead	2	3	4	dead
3	dead	2	3	4	dead
4	dead	dead	dead	dead	5
5	dead	dead	dead	dead	dead
dead	dead	dead	dead	dead	dead

Table 4.1. A transition table for an NFA representing ACCOUNT, incorporating a dead state.

The following description should be read with reference to Table 4.1. The initial partition is made by considering which states are distinguished by ε. This gives the two groups `(1 2 3 4 dead) (5)`. A singleton group cannot be split, so the other group is now considered. The only state which is distinguished by any symbol is `4`, which goes to `5` on `eof`. All other states have out transitions which lead back into the group. The new partition is `(1 2 3 dead) (4) (5)`. This partition cannot be split further. Note that although the input close causes a transition to `(4)`, it does so for *all* members of the group `(1 2 3 dead)`, and so does not distinguish them. After discarding the dead state, we are left with three groups of equivalent states `(1 2 3) (4) (5)`. Together with the start state `firstpos(root)`, this leaves us with a minimized machine with four states.

Construction of the transition function proceeds by choosing a representative from each group (for example `1`, `4` and `5`) and attributing their transitions to the new combined states. In this case, the resulting transition function is as shown in Table 4.2:

start	open	{1,2,3}
{1,2,3}	credit	{1,2,3}
{1,2,3}	debit	{1,2,3}
{1,2,3}	close	{4}
{4}	eof	{5}

Table 4.2. Minimised automaton for ACCOUNT.

Note that this is the same automaton as the one in figure 4.7.

4.7 From Structure Diagram to DFA

This section presents an algorithm, based on one presented by Aho *et al.* (1985), for the direct construction of a state-optimised DFA from a structure diagram. The algorithm involves the calculation of the followpos function as before, but differs in the way it introduces new states. Lewis (1991) and Poo (1991) introduce a new state to represent the current configuration after each leaf node has matched an action. In contrast, the rule used by the direct DFA construction algorithm names a state by binding it to the *set of positions* which can next be matched. This set of positions is a subset (used in the same way as in the subset construction algorithm discussed earlier) of the set of states produced by following the rule for state introduction used by Lewis and Poo. The start state of a process is represented by the set of positions in `firstpos(root)`. Subsequent states and transitions are added by finding all instantiations of the rule

```
TransitionFrom n to followpos(n) on nameof(n),     (where n ∈ S)
.
```

This means that accepting a symbol naming position `n` (where `n` is one of the positions in the set representing the state `S`) will lead to the state named by (the set) `followpos(n)`. During this process, two sets can be identified if they define the same context (that is, they are represented by the same set of positions).

The Algorithm

The method proceeds as follows. The set of deterministic states of the machine under construction will be called `Dstates`. The transition table will be called `Dtrans`. Each element of `Dstates` is a (sub)set of positions, and the start state of the machine is `firstpos(root)`. The accepting states will be those containing the position of `eof`. Figure 4.11 presents the text of the algorithm.

```
procedure subset_struct_diag;
   Dstates := {firstpos(root)};
   while there is an unmarked state T ∈ Dstates do
i)    mark(T);
      for each input ∈ αM do
ii)      inPositions := {p ∈ T | nameof(p) = input};
iii)     U := ⋃ {fset ∈ PN1 | q ∈ inPositions  ∧
               fset = followpos(q)}
         if U ≠ {} and U ∉  Dstates then
iv)         Dstates := Dstates ∪{ U };
         endif;
v)    Dtran[T, input] := U;
      endfor
   endwhile
end subset_struct_diag.
```

Figure 4.11 Construction of a DFA from a structure diagram, adapted from Aho *et al.* (1985).

Consider the application of this algorithm to ACCOUNT. `Firstpos(root) = {1}` and this is entered as a state in `Dstates`. This state is marked (`i`) and becomes the current state under consideration. The algorithm now considers each action in αACCOUNT with respect to the positions which can follow the current state (`ii` & `iii`), in this case, the members of the set `{1}`. Now position 1 matches open, and `followpos(1) = {1,2,3}`, so we know that in the state reached after matching open at position 1, we will be in the state where we can match any of the positions 1, 2 or 3, or:

```
TransitionFrom {1} to {2, 3, 4} on open.
```

`Dstates` and `Dtrans` are now as follows (`iv` & `v`):

```
Dstates = { {1} {2, 3, 4} }
Dtrans = { ({1} open {2,3,4})}
```

Next, $\texttt{Dstate}_{\{2,3,4\}}$ is marked and considered. Position 2 is for `credit`, and the positions which can follow 2 are `{2,3,4}`. This state already exists, so there is no change needed to `Dstates`, but a new transition is added to `Dtrans`:

```
TransitionFrom {2, 3, 4} to {2, 3, 4} on credit.
```

Similar reasoning applies to position 3:

```
TransitionFrom {2, 3, 4} to {2, 3, 4} on debit.
```

`Dtrans` is now

```
{  ({1}      open     {2,3,4})
   ({2,3,4} credit   {2,3,4})
   ({2,3,4} debit    {2,3,4})
}
```

The final position in $\texttt{Dstate}_{\{2,\ 3,\ 4\}}$ is `4`. It is the position for `close` and its `followpos` is `{5}`, which is a new state:

```
      TransitionFrom {2, 3, 4} to {5} on close.
```

The algorithm finally considers $\texttt{Dstate}_{\{5\}}$. This state has no out-transitions. It relates to `eof` and is therefore an accepting state. The algorithm here terminates leaving the following:

```
Dstates = { {1}  {2, 3, 4}  { 5} }
Dtrans =
   {  ({1}         open        {2, 3, 4})
      ({2, 3, 4}   credit      {2, 3, 4})
      ({2, 3, 4}   debit       {2, 3, 4})
      ({2, 3, 4)   close       {5})
   } .
```

Note that this is the DFA given in Figure 4.7.

State Minimisation Properties of Subset Construction

Recall from Chapter 3 that two states are equivalent if they cannot be distinguished by future behaviours, and also that they must not merely be prepared to accept the same symbols, but must accept them in the same positions as well.

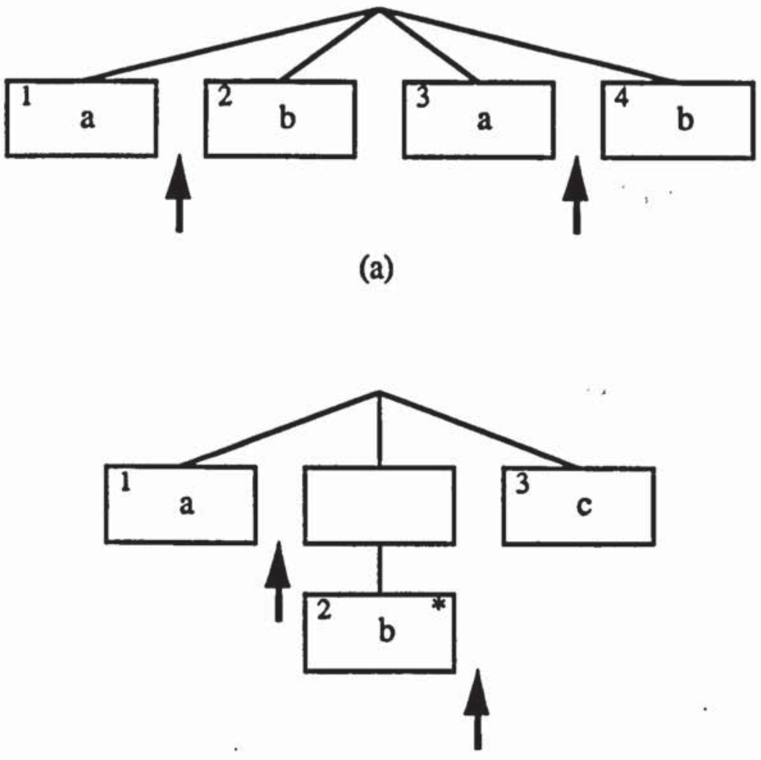


Figure 4.12. Synonymous leaves at different positions.

The two states in Figure 4.12(a) are distinguished by `b`. However, the two states in Figure 4.12(b) are not distinguished by any input.

The optimisation of states which occurs in the structure diagram-to-DFA algorithm arises from the ability to identify as equivalent two states represented by the same set of allowable next positions. If two states can only accept the same symbols (at fixed positions) then they must advance to the same successor state `succ` on a given symbol `s`. Any candidate distinguishing string of the form `<s>'t` (which, by definition, must lead to an accepting state from at least one of the states) must initially take the machine to the same next state `succ` from where to consume `t`. If the machine is deterministic (given), then δ`(succ,t)` must take the machine to a ***unique*** accepting state. So, `<s>'t` must take the machine from both candidate identical states to the same unique accepting state and they can therefore be said to be indistinguishable.

4.8 Summary

This chapter has explained Lewis's (1991) followmap based approach to JSD process transformation in automata-theoretic terms, and has discussed the observed limitations of direct followmap-driven behaviour:

- failure to account for multiple synonymous leaves;
- inability to cope with recognition difficulties (dealt with more fully in the next chapter);
- generation of automata with sub-optimal numbers of states.

With the connection to automata theory established, the chapter showed how these problems can be addressed by applying algorithms for the manipulation of automata. Particularly encouraging is the efficiency gain possible over the direct followmap approach of Lewis.

This discussion has, however, been presented at a level somewhat abstract from *real* JSD process specifications: no mention has been made of the executable operations and extra conditions which can adorn a process structure, the treatment of recognition difficulties has been cursory, and the assumption has been that each leaf node must correspond to the acceptance of an action message rather than representing some computation step internal to the process. So far, therefore, the results of this chapter are only applicable to certain restricted kinds of model processes. The next chapter addresses these real-life considerations and so increases considerably the applicability of these ideas.

Chapter 5

Treatment of Augmented Process Structures

5.1 Observable and Hidden Behaviour of Processes

Previous chapters have explicated and developed the relationship between structure diagrams and automata. In doing so, they have described JSD processes at a rather abstract level; various features of real JSD process specifications have been denied consideration, for example assigned operations and conditions, backtracking constructs and non-reading leaf nodes. This chapter therefore sets out to show how the approach introduced can be extended to cope with process structures augmented with these extra features. The result is a transformational approach which can be applied to most process structures (probably a large enough set not to constrain the applicability of the method), and which can be used to facilitate the automation of desirable transformational strategies (to be discussed in Chapters 6 and 7).

Minsky (1967) describes finite automata as black box machines. That is to say that their internal workings are hidden from their observer, or *environment*. In the simplest case, that of a deterministic recogniser, the environment can tell only two things:

- the contents of the input stream;
- whether or not the machine has accepted an input string.

Although FAs can be described in terms of transition graphs which explicitly represent states, the existence of anything corresponding to a state in the workings of the machine can be infered only from the machine's observable behaviour. Certainly, nothing can be said about the structure of the machine's internals or the configurations of this structure which correspond to the things the environment regards as states. Indeed, as was noted in Chapter 3, this lack of structural bias is a great attraction of automata as modelling devices. It follows that there may be changes in the internal configuration of an FA which are completely hidden from its environment. This is certainly the case with the software simulations of automata which have been presented in previous chapters, in which the internal configuration of these machines includes storage locations holding binary code, a program counter and the miscellaneous variables needed to implement the abstract machine. The environment of such a machine (for example the syntax analyser of a compiler which is calling a DFA-based scanner for the next program token) is completely unaware of the updates made to, say, the program counter while the machine executes.

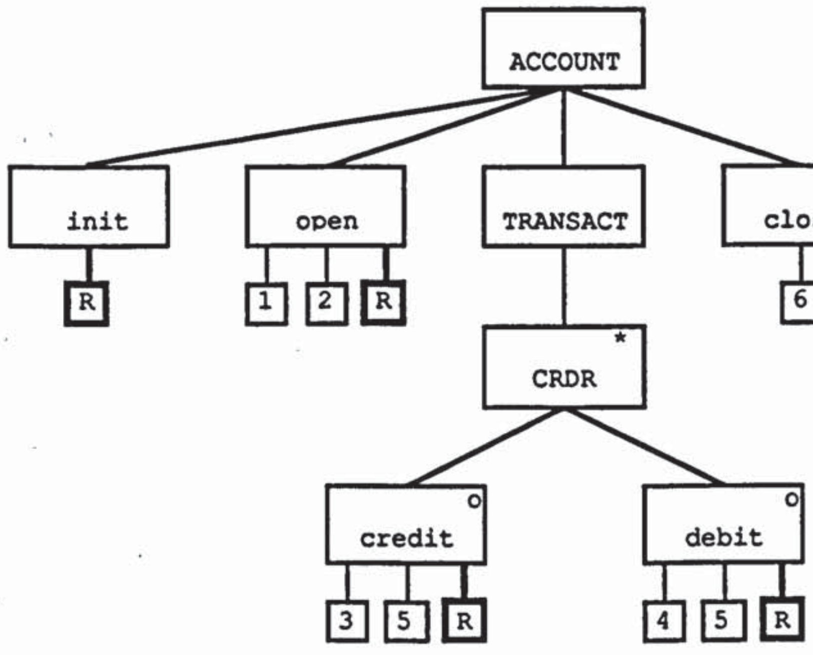
The great emphasis placed by JSD on the modelling of events, and their tracking by what are essentially finite state machines, is motivated by the desire to build a stable framework with which to organise and coordinate the provision of functionality. Most of the computation which contributes to the functionality of a JSD system is hidden beneath the automaton abstraction, and is specified in terms of executable operations together with various controlling conditions which the developer assigns to process structures. This chapter is concerned with this internal behaviour of JSD processes, and particularly with how it relates to the external appearance of these processes when viewed as automata. As will be seen, functional code to perform database updates and output, say, can be added without affecting the integrity of a process as a DFA. Control predicates are another matter and are not really compatible with the DFA abstraction. An approach to the partial transformation of process structures into automata is proposed which can, under certain circumstances, allow arbitrary conditions to exist within a valid automaton abstraction.

It has been seen that any NFA can be transformed into a corresponding DFA in such a way that choices among branches for accepting a given symbol can be tracked in parallel. There may, however, be times when the hidden behaviour associated with a branch is different from the hidden behaviour associated with an alternative so that parallel simulation is not possible. In such cases, subset construction alone is insufficient to ensure that the appropriate hidden behaviour occurs when it should. This is a new view on the problem of recognition difficulties (Jackson, 1975), and the change in perspective facilitates a new solution, which is described at the end of the chapter.

5.2 Executable Operations

Recall that JSD model processes (essentially, executing automata) track their corresponding entities and can therefore be inspected to yield information about the state of the real world. However, the finite nature of these modelling devices limits the information which can be captured by the notion of a state as formalised in Chapter 3, that is, as a named equivalence class of behavioural histories. JSD process specifications are therefore typically elaborated with an additional layer, comprising entity and action *attributes* together with associated update operations. The resulting two layer description helps the JSD developer to maintain a separation of concerns between the dynamic behaviour of the system over time, which is rooted in a relatively stable aspect of the problem domain, and the update logic, which maintains detailed status and summary information in support of the intended system's (volatile) functional requirements.

Structure diagrams which have been augmented with update operations will be denoted by their name together with the subscript '$_{ops}$'. Figure 5.1 presents an augmented structure diagram for ACCOUNT$_{ops}$.



```
OperationTable
R →  read(input);
1 →  balance := 0;
2 →  num_trans := 0;
3 →  balance := balance + input.amount;
4 →  balance := balance - input.amount;
5 →  num_trans := num_trans + 1;
6 →  write(balance);
```

Figure 5.1 An augmented structure diagram for ACCOUNT$_{ops}$

The values required to update process attributes are supplied as attributes of the action messages in the process's input stream (e.g. `input.amount`). The question of which attributes to record and update is the subject of extensive discussion by Cameron (1988) and is also dealt with in detail by Jackson (1983).

The subset construction approach requires that operations be assigned only to leaf nodes. It is accepted practice (Jackson, 1975; Cameron, 1988) to assign operations to interior nodes with the understanding that they are to be sequentially composed with the other descendants of that node. This is purely a notational shorthand to avoid cluttering diagrams with sequence node (in the above example, it would avoid the introduction of the `_init` node as a placeholder for the first read operation). There is no loss in expressive power as a consequence of restricting operation assignments to leaf node, and any notational inconvenience is easily mitigated by the use of a suitable diagram editor such as that provided by the PRESTIGE JSD workbench (Lewis, 1991).

Representing Structures Augmented with Assigned Operations

The operations which can be performed on the attributes of an entity can be represented by an `OperationTable`. For now we model this as a straightforward mapping:

```
OperationTable = map op_index to operation
   where
      op_index = N1 .
```

In the PRESTIGE project (Bass *et al.*, 1991) a more sophisticated representation of `OperationTable` was chosen. This was used as the basis of the specification of the textual substitutions necessary to realize the inversion transformation in a conventional way (i.e. using goto statements) and is discussed in Chapter 7. Figure 5.1 shows the `OperationTable` for ACCOUNT_{ops} in terms of the simple mapping representation.

The sequence of operations to be performed when accepting an action is termed an *action body*, after Poo (1991). Poo, and also Lewis (1991), take the view that the action body to be executed at any particular time is a function of the name of the action being received. However, this gives rise to problems when there are multiple synonymous leaves, and so it is better to regard the choice of action body to be a function of the in-order *position* of the leaf node at which the action is received. This can be modelled as

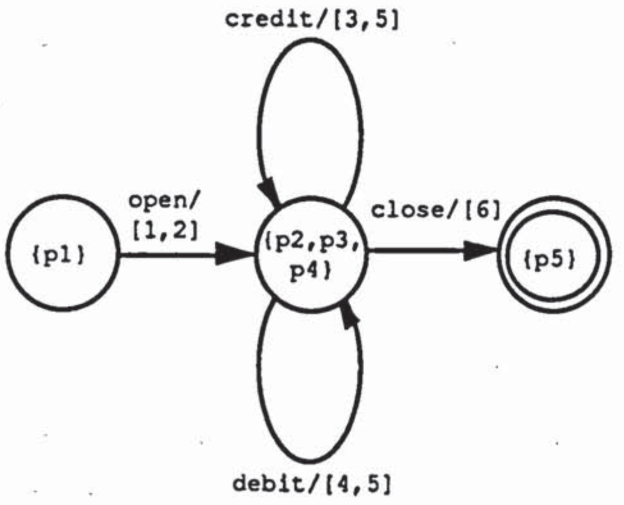
```
operation_map = map position to action_body
   where
      action_body = seq of N1   .
```

It is possible to tabulate the `operation_map` for ACCOUNT_{ops} in Table 5.1.

position	action body
p1	[1,2]
p2	[3,5]
p3	[4,5]
p4	[6]

Table 5.1. An `operation_map` for ACCOUNT_{ops}.

With the constructs specified above, it is possible to define finite automata augmented with assigned operations as a six-tuple $\{S, \alpha M, \delta, \gamma, F, q_1\}$ where γ: `state` $\times$ `input` $\rightarrow$ `actionbody`, and other fields have the same meaning as previously. Figure 5.2 presents a graphical view of a DFA_{ops} for ACCOUNT, together with an *effects table* for γ (recall that the states of a DFA are here modelled as **set of** `position`). In this diagram, transitions are labelled *action/actionbody*.



γ	open	credit	debit	close
{p1}	[1, 2]	–	–	–
{p2,p3,p4}	–	[3,5]	[4,5]	[6]
{p5}	–	–	–	–

Figure 5.2. A DFA for ACCOUNT_{ops}, together with an effects table for γ.

Note the similarity with the output tables of GSMs discussed in Chapter 3. Clearly, if all the assigned operations are write statements, then the similarity is exact. Where these operations are assignments, then such an automaton can still be regarded as GSM-like providing that the results of these assignments are not used to influence subsequent control flow. This point is covered in more detail in Section 5.3.

Constructing an Effects Table

Transformation of an augmented structure into a corresponding automaton depends on correctly constructing the effects table for γ. This can be achieved by an enhancement to the subset construction algorithm, making use of the one-one relationship between positions and action bodies. The new algorithm is given in Figure 5.3. The additional lines required to construct γ (stored in the variable `Dops`) are presented in italics and are marked `(i)` & `(ii)`. In line `(i)`, `p` is matched to the singleton element of the set `inPositions`. In line `(ii)`, the expression `action_body(p)` yields the action body assigned to the leaf node at position `p`.

```
procedure dfa_ops;
   Dstates := {firstpos(root)};
   while there is an unmarked state T ∈ Dstates do
      mark(T);
      for each input ∈ αM do
         inPositions := {p ∈ T | nameof(p) = input};
```

```
             U := ⋃ {fset ∈ ℙN1 | q ∈ inPositions  ∧
                  fset = followpos(q)}
             if U ≠ {} and U ∉  Dstates then
                Dstates := Dstates ∪{ U }
             endif;
             Dtran[T, input] := U;
i)           inPositions = { ?p };
ii)          Dops[T, input ] := action_body(p)
          endfor
       endwhile
end dfa_ops.
```

Figure 5.2 A Structure$_{ops}$ to DFA$_{ops}$ transformation algorithm.

The action body to be performed during a state transition is a function of the position which is matching the current action. The number of action bodies is equal to the number of leaves of the tree. As we have seen, however, the number of states of a minimum state DFA is usually less than the number of leaves of a corresponding structure. The above algorithm extends Lewis's (1991) work by recognising that it is possible to perform different action bodies and still get to the same (general) state. This follows if it is remembered that, in general, different positions can be matched on the way to the same state (and indeed this explains the state-reducing effects of the subset construction algorithm over an approach based on direct construction from a follow map).

A simple modification to the algorithm for DFA animation presented in Chapter 3 provides for the execution of operations during the recognition process. The algorithm is presented below, with the new line in bold:

```
state := $q_1$;
while not end of input do
   read(input);
   perform( γ(state,  input)  );
   state := δ(state, input);
endwhile;
if state ∈ F then write(1) else write(0);
```

Assignment of Operations to Null Nodes

The structure diagram notation allows the use of *null* leaf nodes as components of two-way selection. For example, the structure in Figure 5.4 is equivalent to the regular expression a(x|ε)b. A look at the available literature on Jackson methods reveals no clear answer to the question: "Can executable operations be assigned to null nodes?". Lewis's software tool (1991) forbids the assignment of operations to null nodes to allow his follow set based approach to function. Consider the structure in Figure 5.4.

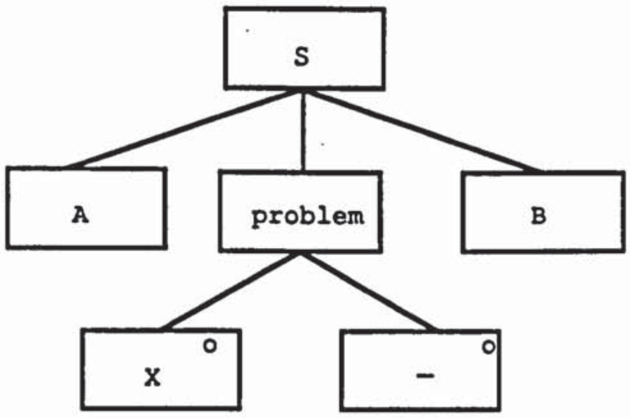


Figure 5.4. A structure with null components.

All sources agree that the condition for execution of the leaf labelled x is that an x message has been read in by the read ahead positioned at the end of A. However, there is no clear guidance about what happens if any other type of message is obtained. The two possible codings are as follows:

```
View 1
read(msg)
A seq
   A;
   read(msg);
A end;
Problem sel msg = X
   X;
   read(msg);
Problem end
B seq
   B;
   read(msg);
B end;


View 2
read(msg)
A seq
   A;
   read(msg);
A end;
Problem sel msg = X
   X;
   read(msg);
Problem alt msg <> X
   Xabsent;
   -- to read or not to read?
Problem end
B seq
   B;
   read(msg);
B end;
```

The following arguments seem convincingly to legislate against the allocation of *read* operations to null components. They address three points of view: practical, JSD-conceptual and language-theoretic. From a practical point of view, if the `Xabsent` branch in the second fragment above has operations allocated to it, then none of these operations can read. If they do, then clearly the standard read-ahead is violated. Conceptually, null nodes should not read since they do not correspond to any action occuring in the real world. In language-theoretic terms `(x | ε)` is saying "a possible x". It is not saying "either an x or anything else". For example, the regular expression `a (x | ε) b` admits the strings

```
<a, b> , and
<a, x, b> ,
```

but nothing else. A null component does not therefore match (and therefore cannot read) "any action other than X" i.e., it does not mean `not(X)`.

No such clear argument can be made against the allocation of non-reading operations to null components. However, the present approach does not allow such allocations due to its underlying reliance on follow sets. The rules for follow sets (see Figure 4.1) prevent a null node from appearing in the follow set of any other node in a structure (null nodes are effectively skipped over). Therefore, such a node will not be visited during follow set based execution, so any operations assigned to it will be unreachable. This constraint does not cause a loss of expressive power, however, as it is possible to arrange for operations to be performed in the absence of a particular message using *backtracking*. (Backtracking is discussed in Section 5.4.).

5.3 Dealing with Conditions

It is necessary to distinguish two kinds of condition which arise in JSD processes. Firstly, there are predicates on the type of the current input message. These are the conditions represented in transition tables of finite automata. Secondly, there are predicates which refer to *data values*: either message attributes or entity attributes, or both. An example from a banking application might be

```
(balance - msg.amount) < 0 ,
```

which specifies the condition under which a withdrawal of funds would cause an account to go into overdraft.

As far as the first kind of condition is concerned, the present approach automatically generates them in the process of conversion from structure diagrams to DFAs. This represents an improvement over Lewis's (1991) approach which requires that such conditions be provided through human intervention. The second kind of condition is much more problematic, because instances violate the integrity of the finite automata model: no longer is a process's state-changing behaviour influenced only by the type of messages in its input stream, but also by potentially obscure conditions on data values. Some examples of behaviours which required conditions on process or message attributes follow.

Bounded iterations

Consider the popular library BOOK example from the JSD literature (Cameron, 1986; Poo, 1991) for which a structure diagram was given in Figure 3.7. The library management might wish to enforce a constraint that a book can be renewed only twice before it must be returned and made available to another borrower. Strictly, the only way to represent this with any regular structure is by enumerating all possible sequences of renews. For example:

```
loanpart = lend (ε | renew | renew renew) return       .
```

Clearly this is rather clumsy, and would soon become unworkable for larger upper bounds on the iteration. A solution is to break with the pure metalanguage of regular expressions and specify loan part as

```
lend renewⁿ return   (0 ≤ n ≤ 2)    .
```

The simplest implementation of this scheme involves adding a condition directly to the iteration of renew, giving rise to code of the form:

```
read(msg);
lend;
renew_count := 0;
read(msg);
while (msg <> return) and (renew_count < 2) do
   renew;
   renew_count := renew_count + 1;
   read(msg);
endwhile;
return.
```

There is an objection to this kind of coding in model processes, which is that the condition on the loop affects control in a way not captured purely by the syntactic structure of the process. Cameron (1989) proposes delegating responsibility for ensuring that constraints of this sort are observed to the input subsystem. In his scheme, a context filter would keep a count of the number of renew messages and would only pass a maximum of two such messages on the the model, whose loanpart would remain specified as

```
loan renew* return   .
```

Clearly the context filter will still need a condition on the number of `renews` to perform its task, but at least the syntactic purity of the model is preserved.

Restrictions on the range of attribute values

Consider the following constraint on the use of a bank account: any attempted `debit` which takes the account into overdraft is not to be allowed. This constraint refers to an attribute of the `debit` message, the `amount`, and an attribute of the entity, the `balance` of account. Unless the alphabet of the process is changed so that `debit` is split to handle the two cases (say into `OKdebit` and `NOKdebit`) there is no way to represent such a constraint using regular structures. A control predicate must be added to test explicitly the attribute value. One possible implementation fragment would then be:

```
while msg <> close do
   if msg = credit then
      credit;
      read(msg);
   elseif msg = debit then
      if balance - msg. amount >= 0 then
         debit;
      endif;
      read(msg;
   endif;
endwhile;
close;
```

Note that the prevention of overdrafts cannot be coded as

```
elseif msg = debit and balance - msg. amount >= 0 then
   debit;
   read(msg);  -- only executed when debit is ok!!
endif;  ,
```

because the read-ahead requirement would be violated if there was indeed an attempt overdraw. The correct solution, the one presented first, features an action body which incorporates control structure. Such *structured action bodies* are discussed in detail later in this section.

Allowing Data-conditions without Violating the Automata Abstraction

Application of the read-ahead convention as applied to model processes gives rise to structures in which all leaves are held to read. In many function processes there will be non-reading leaf nodes which specify hidden behaviour. Distinguishing these *hidden* nodes from reading leaf nodes makes it possible to propose an optimisation to the `Structure-to-DFA` transformation developed in Section 4.7 which may be useful in certain circumstances. Hidden nodes occur when action bodies must be structured in some way so that iterations or selections, controlled by attribute values, must be negotiated in the (conceptually atomic) transition between one general state and another. The straightforward implementation is to treat all leaf-nodes the same (that is, as precursors of general states) whether they correspond to external events or not, and just have execution pass through non-reading 'states' until a true general state is reached. Lewis (1991) shows how this can be done. As he notes, however, this approach can be somewhat inefficient, as it ignores opportunities to use the standard programming control structures, relying instead on a control variable which must be explicitly tested. Consider a sequence of hidden leaves equivalent to the regular expression `(a b c)`. The natural implementation of this sequence is

```
A; B; C;
```

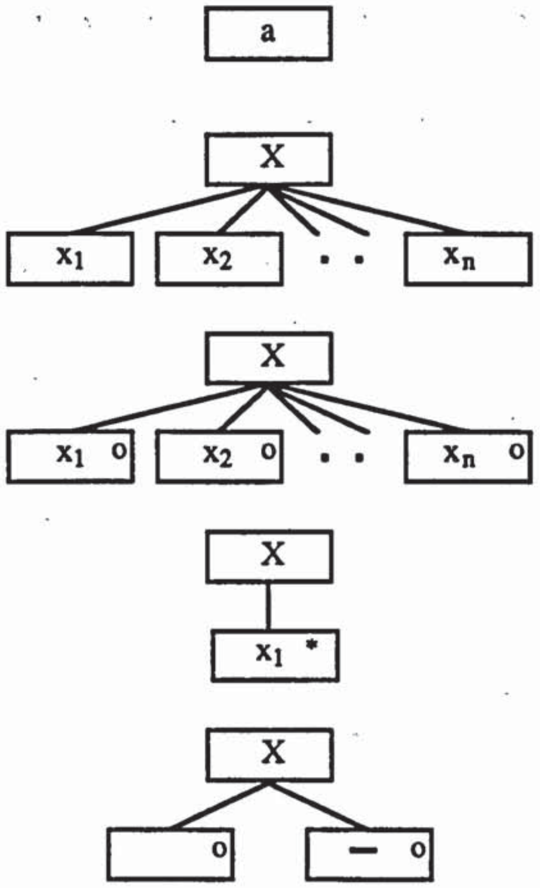
In the worst case, Lewis's approach could give rise to the following implementation:

```
while state <> eof do
   case state of
      Cstate => C; state := eof;
      Bstate => B; state := Cstate;
      Astate => A; state := Bstate;
   endcase;
endwhile;
```

Here, control information is held in the variable `state`, which must be explicitly tested in order to determine the next processing component to be executed. It is tempting to wonder whether or not such a baroque scheme can be avoided, and if so under what circumstances. The present approach to this issue depends on the introduction of the notion of an *action structure*. The sequence `(a b c)` is a simple example of an action structure, although most such structures would also contain iterations and selections.

Definition of Action Structures

It is first necessary to define the function `Last: StructureDiagram → setof action`. This function returns the set of actions which can end a string in the (sub)language represented by the diagram. Rules defining `Last` are given in Figure 5.5. These are symmetrical with the rules for the function `First` given in Chapter 4. Aho *et al.* (1985) give similar rules for *binary* trees representing regular expressions.



a is a leaf node
`Last(a)` = `{a}`

X is a sequence node
`Last(X)` = `Last(`x_n`)`

X is a selection node
`Last(X)` =
`Last(`x_1`)` ∪ `Last(`x_2`)` ∪..∪ `Last(`x_n`)`

X is an iteration node
`Last(X)` = `Last(`x_1`)`

X is a nullable selection node
`Last(X)` = `{`x_1`}`

Figure 5.5. Rules for the computation of the `Last` function

An action structure is a Jackson tree which is executed on receipt of a single action. Clearly all reading leaf-nodes can be considered as base case action structures. More generally, any tree T is an action structure iff

> All the nodes in `Last(T)` are reading nodes, and all other leaf nodes (`Leaves(T) \ Last(T)`) are not.
>
> (`Leaves(T)` returns a set containing the leaf nodes of T.)

The leaf nodes of model processes give rise to very simply structured action bodies (i.e., singleton sequences), but more complex action structures give rise to equally complex action bodies. A change in the representation of `OperationTable` (see Section 5.2) caters for this modification:

```
    OperationTable = map position to ActionStructure  .
```

To maintain consistency with these changes, the root of an action structure has to be labelled (by a tool user) as such, so that it can be treated as a position by the subset construction algorithm.

An example: A Banking System Statement Lister

Consider the requirement to list the transactions undertaken on an ACCOUNT, one per line (in addition to a header and footer lines). The ACCOUNT process of Figure 5.1, could be elaborated to write a stream of messages summarising its activities. The STATEMENT-LISTER process will read this stream, which will have the structure `open (credit | debit) close`. The user requires STATEMENT-LISTER's output to have a slightly different structure: `header (cr-line | black-debit-line | red-debit-line) footer`. The decision as to whether to print a `black-debit-line` or a `red-debit-line` depends on the value of the `balance` field in the incoming messages in the obvious way. Figure 5.6 shows the process specification of STATEMENT-LISTER.

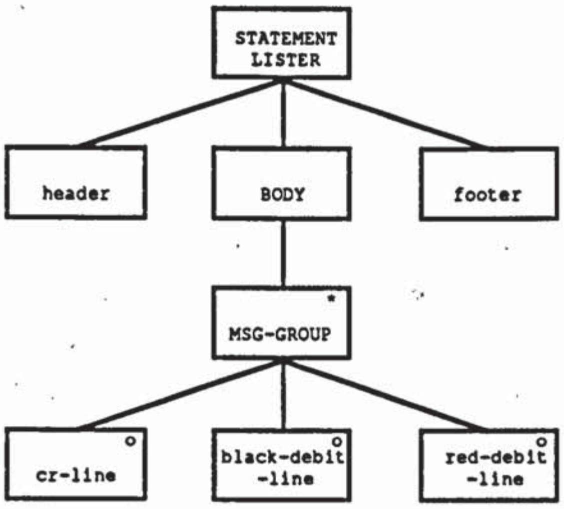


Figure 5.6. Process structure of STATEMENT-LISTER.

To implement this as an automaton, `msg-group` is nominated the root of an action structure, AS_1, a selection of the three components `(cr-line | black-debit-line | red-debit-line)`. AS_1 incorporates the conditions required to control its execution. Subset construction is performed on the subtree whose leaves are `{header, msg-group, footer}`. Transition and effects tables for the resulting automata are shown in Figure 5.7.

δ	header	msg	footer
1	2	–	–
2	–	2	3
3	–	–	–

γ	header	msg	footer
1	action body(1)	–	–
2	–	AS_1	action_body(3)
3	–	–	–

Figure 5.7. Transition and effects tables for STATEMENT-LISTER.

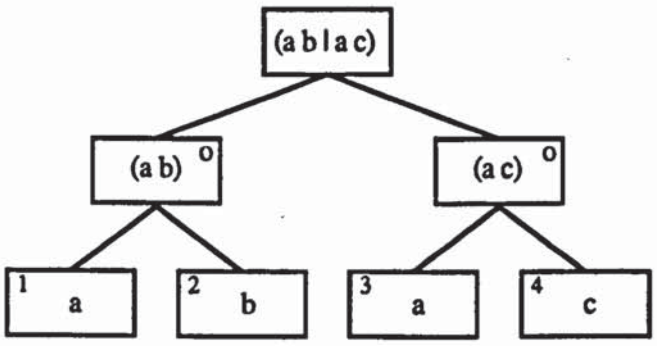
The advantages of the action structure approach to dealing with data conditions can be stated as follows:

- It gives rise to more efficient implementations than the control-variable method as it involves less explicit testing of the control variables's value;
- It allows better congruity with the finite automaton abstraction, as each 'state' still relates just to input history;
- The transformation algorithm has less states to consider and so runs more quickly.

Further aspects of this approach, connected with interprocess communication, are discussed in the next chapter.

5.4 Recognition or Parsing Difficulties

After presenting the basic JSP method, Jackson (1975) addresses a class of problem called *recognition difficulties*. A recognition difficulty arises when there is uncertainty about what a process should do next based on consideration of only the next symbol in the input stream. A simple regular expression which describes a language with this property is `(ab|ac)`. Figure 5.8 shows the corresponding structure diagram. A recogniser for this language which can look only one character ahead has no way of knowing whether a string starting with an `a` is the prefix of `<a,b>` or `<a,c>` (because the two positions in `firstpos(root)`, i.e., `{1,3}`, have the same name). It has been shown that for any NFA there exists a corresponding DFA which recognises the same language, and a clue as to a suitable DFA for this example can be obtained by factoring the expression so: `a(b|c)`. However, such factoring will not address recognition difficulty problems where the action body to be executed is dependent on which of the similarly-named branches is taken, and, unfortunately, it is in precisely such cases that recognition difficulties arise in JSD.



position	action body
p1	[1,2]
p2	[3,4]
p3	[5,6]
p4	[7.8]

Figure 5.8. Structure diagram for (a b | a c) together with possible operation map.

Standard Jackson Solution to Recognition Difficulties

In the standard JSD approach to recognition difficulties, the developer nominates or *posits* one of the branches as the one to assume is correct using a special kind of two-way selection (the assumption is that there will only ever be two alternatives, although in general there could be arbitrarily many) . Execution starts down this branch. If it transpires that the assumption was wrong, then this branch is abandoned and the second branch, the *admit* branch, is followed. Two important items of knowledge are needed to realise this *backtracking* behaviour:

1) the state which it transpires that the process *should* be in;
2) the sequence of operations which have been performed in error, and the alternative sequence which should have been performed instead (so that the correct updates to attributes are made).

Below is a standard implementation of backtracking to solve the recognition difficulty expressed in Figure 5.8.

```
read(msg);
-- posit a b (i.e., match position 1)
perform([1,2]);
read(msg);
if msg = c then goto admit endif;
perform([3,4])     -- match position 2
goto out;
admit: - a c
   undo(actionbody(1));
   perform([5,6]);    -- match position 3
   perform([7,8]);    -- match position 4
out:
```

A disadvantage of this implementation is its reliance on the `goto` statement. Use of the `goto` in the realisation of backtracking prevents the use of languages such as occam™ and Smalltalk™ which lack such commands. Also, it leaves the program in a state where it is much less amenable to further transformation. Finally, despite exhortations by Jackson (1983), Cameron (1986) and others that transformed code should be regarded as *object code* and therefore that the utilisation of `gotos` for implementing techniques such as backtracking and inversion should not be deprecated, there remains considerable commercial resistance to the use of JSD because of its use of `goto` statements in realising its transformations (Cameron, 1989).

Possible DFA Solution

Consider the NFA arising from the structure shown in Figure 5.8. The transition diagram for this machine is shown in Figure 5.9. Here, if the machine receives an initial `a` which turns out to be a prefix of `<a,b>`, then it is required that the actions `[1,2]` be performed. On the other hand, should the input string turn out to be `<a,c>`, then the correct actions are `[5,6]`.

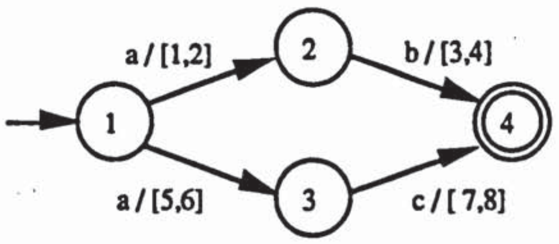


Figure 5.9. An NFA for `(a b | a c)` posing a recognition difficulty.

An interesting way of viewing a backtracking solution to this recognition difficulty is shown in Figure 5.10. The assumption is made that the initial `a` is a prefix of `<a,b>`, and the operations `[1,2]` are performed. As these operations may have undesirable side-effects on process attributes in the event of the assumption being proved wrong, a snapshot of the process's attribute values is taken prior to execution of the operations. Once in state `{2,3}`, a deterministic choice is made based on the value of the second input symbol. If it is a `b`, then the posited assumption was correct and execution can continue normally. If on the other hand it turns out to be `c`, then the initial values of the attributes must be restored, and the full sequence of operations for `<a,c>` performed.

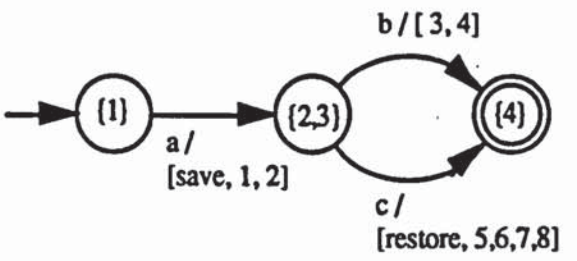


Figure 5.10 A recognition difficulty solution.

The use of `save` and `restore` operations to manage the entire state-vector of a process seems rather clumsy and indiscriminate. Firstly, some attributes may be unaffected by the wrongly-performed operations. Secondly, there are some circumstances where the side-effects may be either *neutral* or *beneficient* (Jackson, 1975) from the point of view of the admit branch, and may therefore be ignored. These issues can be considered independently of present considerations, and the reader is refered to Jackson (1975) for further details.

The approach presented here avoids the use of `goto` statements. This offers two major advantages:

1) it allows backtracking to be realised in languages which lack a `goto` facility;
2) it yields code which is amenable to further transformation.

The first of these advantages extends the understanding of the realisation of JSD specifications in object-oriented environments provided by the work of Lewis (1991). The second increases the generality of the transformational approach being developed in this thesis.

A Subset Construction Algorithm for Backtracking Problems

Recall that subset construction deals with nondeterministic problems by building a DFA which keeps track of all the possible states a corresponding NFA might be in at any particular time. This algorithm already builds the correct transition table for processes with recognition difficulties. However, the effects table will not be constructed correctly. At the point where a recognition difficulty occurs, subset construction will build an arc to a state represented by a subset containing two positions of the same name. The algorithm will arbitrarily assign the action body of *one* of these two positions (the first one it finds, which depends on the implementation of the set abstraction) to this transition. This action body will then be executed, regardless of which of the two positions is actually supposed to be matched by a given input. Where the previously presented subset algorithm treats all positions in a subset equally, it will now be necessary to partition the subset into two groups: those positions which can follow in the posited branch, and those which can follow in the admit branch.

The new algorithm is presented incrementally. Firstly, we ignore the problem of building the effects table and just concern ourselves with building the transition table. The first increment, then, is equivalent to the subset algorithm of Aho *et al* (1985) presented in Figure 4.11. The difference is that the inner loop of the algorithm, which considers the transitions from the state under consideration on each possible input symbol, now has to deal with two discriminated cases:

1) when a recognition difficulty is detected;
2) when there is no such difficulty.

In outline, the new algorithm has the following structure:

```
Dstates := {firstpos(root)};
while there is an unmarked state T ∈ Dstates do
   mark(T);
   for each input ∈ αM do
      inPositions := {p ∈ T | nameof(p) = input};
      U := ⋃ {fset ∈ PN1 |
                  q ∈ inPositions  ∧  fset = followpos(q)}
      if |inPositions| > 1 then
         "Handle recognition difficulty"
      else
         "No recognition difficulty"
      endif;
      if U ≠ {} and U ∉  Dstates then
         Dstates := Dstates ∪{ U };
      endif;
      Dtran[T, input] := U;
      Dops[T, input ] := action_body(p);
   endfor;
endwhile.
```

A new data structure, the `PositTable` (`state` → `position`) is introduced (line `(ii)`), the purpose of which is to record the `Nstate` (see Section 4.5) to which it is posited that a particular input will lead starting from a given state. This table will be sparsely populated, with only one entry per recognition difficulty per process.

Figure 5.12 presents a refinement of the above outline.

```
Dstates := {firstpos(root)};
admitseq := [restore];
while there is an unmarked state T ∈ Dstates do
   mark(T);
   for each input ∈ αM do
      inPositions := {p ∈ T | nameof(p) = input};
      U := ⋃ {fset ∈ PN1 | q ∈ inPositions  ∧
fset = followpos(q)}
      if |inPositions| > 1 then
         -- Handle recognition difficulty
(i)      identify positPos and admitPos;
(ii)     positTable(U) := followpos(positPos);
(iii)    effect := [save] ++ actionbody(positPos);
(iv)     admitseq := admitseq ++ actionbody(admitPos);
      elsif |inPositions| = 1 then
         -- No recognition difficulty
(v)      inPositions = { ?p };
(vi)     effect := actionbody(p);
         if p ∉ positTable(T) then
            effect := admitseq ++ effect;
            admitseq := restore;
      endif;
      if U ≠ {} and U ∉  Dstates then
         Dstates := Dstates ∪{ U };
      endif;
      Dtran[T, input] := U;
      Dops[T, input ] := effect;
   endfor;
endwhile;
```

Figure 5.12. A subset construction algorithm for the conversion of augmented structures into DFAs which allows recognition difficulties

The following notes refer to the labelled lines in Figure 5.12:

i) Either the user must be asked to nominate the posited position or else one must be chosen by a default rule, say, "pick the lowest numbered position".

ii) Record that the state `U` is posited to be the state which would be reached by matching the input to `positPos` starting from state `T`.

iii) Ensure that the state vector will be saved when the posited branch is followed.

iv) The `admitseq` is made up from the actionbody of `admitPos`.

v) In this case, there is no recognition difficulty, so the singleton element of `inPositions` is the position which is to match the input.

vi) The `positTable` is consulted to find the set of positions which it is posited that are to be matched next. If the next input is not matched by any of these positions, then backtracking to the admit branch will be necessary. The `admitseq` then represents the desired effects table entry.

An example of the algorithm's operation on the recognition difficulty presented in Figure 5.9 is now presented.

The first state added to `Dstates` is `firstpos(root) = {1,3}`. This state is marked and considered as the first value of `T`. As `name(1) = name(3) = a`, `inPositions = {1,3}` when `input = a`. `U` is then the union of the followpos of each element of `inPosition`, that is, `{2,4}`. As `|inPositions| > 1`, the new recognition difficulty branch is entered. Assuming the user nominates `1` as the `positPos` and `2` as the `admitPos`, then the following assignments will be made:

```
PositTable[{2,4}] := 2; -- i.e. posit that an 'a' matches position
2
Dstates := Dstates ∪{{2,4}}; -- add to algorithm
Dtran[{1,3}, a] := {2,4};
effect := [save,1,2];
admitseq := [restore,5,6];
```

Next, the state `{2,4}` is marked and becomes the value of `T`. There is no (new or continuing) recognition difficulty so the lower alternative for the consideration of inputs is taken and the existing difficulty resolved. Only inputs `b` and `c` give rise to transitions. Consider `b` first of all. If a `b` is obtained, this constitutes satisfactory resolution of the posit branch, as in this case, `U = {eof}` and `p = 2`, and the `PositTable` records that in state `{2,4}` the normal execution of the posited branch continues by matching with position `2`. The following assignments can be made:

```
Dstates := Dstates ∪{{eof}};
Dtran[{2,4}, b] := {eof};
Dops[{2,4}, b]  := [3,4];
```

Now, consider `c`. In this case, `U = {eof}` and `p = 4`. The `PositTable` records that in state `{2,4}` normal execution proceeds by matching position `2`, but `c` matches position `4`. Therefore, if a `c` is obtained at this point, this means that the posit was inappropriate and that the admit branch should have been executed. The following assignments can be made:

```
Dtran[{2,4}, c] := {eof}; -- note that {eof} is already in Dstates
Dops[{2,4}, 2]  := [restore, 5,6,7,8];
```

This algorithm therefore constructs the solution shown in Figure 5.10. Appendix I gives a Smalltalk-80 implementation of this algorithm.

5.5 Conclusion

The subset construction approach introduced in the previous chapter has been extended to incorporate various features which JSD uses to augment structure diagram specifications. Operations have easily been accommodated as the definition of finite automata does not forbid the hidden activity they imply. Conditions were shown to be rather more problematic as they affect the external determinism of processes. By introducing the idea of action structures, internal conditional behaviour has been facilitated, together with the ability only partially to transform structures into automata (typically leaving lower levels in the tree in their standard form). The approach supports the generation of transformationally derived code which is both efficient and clear.

Subset construction has been used as the basis for an algorithm to generate DFAs which handle recognition difficulties. This transformation, in common with the others presented in the thesis, does not rely on `goto`-statements, and therefore its use is possible with languages such as occam and Smalltalk.

As a result of these enhancements to subset construction, a transformational approach has been obtained which is general in its applicability and which produces efficient implementations. The ability to transform the majority of JSD processes into DFAs makes it possible to automate some very desirable *global* implementation strategies. This is the subject of the next chapter.

Chapter 6
Applications

6.1 Introduction

Previous chapters have developed a method for automatically transforming structure diagrams into finite automata. The ability to effect this transformation is useful to the JSD developer because the DFA representation supports a variety of useful applications. While the static representation of the syntax of observable behaviour provided by structure diagrams seems to provide a useful source of insights to the modeller (Renold, 1988b), a dynamic model of behaviour, as offered by finite automata, appears helpful when the attention of the developer turns to implementation considerations. The latter representation potentially offers support for activities such as implementation, validation and performance prediction. The present work has focused on implementation issues, but some suggestions of relevance to the other areas are made in Chapter 8.

6.2 Inversion

The standard Jackson implementation of the inversion transformation (see Chapter 3) involves a text-pointer variable (stored in the state vector, and often named qs), which enjoys a similar relationship with the read points of a process as the 'state' of a corresponding automaton. This variable can be manipulated like a state variable to allow the suspension and resumption of a process's execution as required. Below is shown the text of ACCOUNT inverted with respect to its input stream, in an obvious pseudocode. The input message and the state vector of the appropriate instance of ACCOUNT are passed to the routine as parameters.

```
procedure account_inverted(input: msg_type; sv: sv_type);

dispatcher case sv.qs of -- jump to just past last suspend point
   1 => goto L1;
   2 => goto L2;
   3 => goto L3;
   4 => goto L4;
   5 => goto L5;
dispatcher end;
L1:
open seq
   actionbody(open);
   sv.qs := 2; return; L2: -- such lines replace 'read(input)'
```

```
open end;
transact itr while input <> close
   crdr sel input = credit
      credit seq
         actionbody(credit);
         sv.qs := 3; return; L3:
      credit end;
   crdr alt input = debit
      debit seq
         actionbody(debit);
         sv.qs := 4; return; L4:
      debit end;
transact end;
close seq
   actionbody(close);
   sv.qs := 5; return; L5:
close end;

end account.
```

The following points can be made about this text:

- Although it has been been allowed in the code above, many programming languages (for example, Pascal and Ada) do not permit jumps into control blocks. In such cases it is necessary to 'flatten' iterations and multiway selections using conditional `goto`-statements so that a completely destructured text is produced (a transformation rule for the flattening of `while`-loops was given in Section 2.3.).
- The initial `case` statement (labelled `dispatcher`) relates integer values of the current text pointer (`sv.qs`) of the process to `goto`-destination labels in the process text.
- As regards the control indirection provided by the `gotos` in the dispatcher, an approach will shortly be illustrated which allows all the substantive code of a process to be embedded directly into a `case` statement;
- In cases where an implementation language fails to offer a `return` statement, the effect can be simulated by inserting a jump to the end of the text of the routine.

An alternative to the style of inversion coding illustrated above is to explicitly manipulate the program counter of the computer (real or virtual) on which an implementation is running. Lewis (1991) illustrates an ingenious implementation of this kind in Smalltalk-80, which very unusually for a high level language affords direct read-write access to the program counter of the virtual machine (Goldberg & Robson, 1983). This option is not, however, generally available to the JSD implementor.

Each invocation of an inverted procedure can be thought of as representing the transition between one general state and the next, and can be seen as the execution of a single action body. Given a DFA representation, it is straightforward to implement inversion because the relationship between states, inputs and action bodies is explicit. Below is shown the text produced by applying this approach to ACCOUNT:

```
procedure account_dfa_inverted(input: msg_type; sv: sv_type);
state renames sv.state;
   case state of
      1 =>
         case input of
            open =>
               actionbody(open);
               state := 2
         endcase;
      2 =>
         case input of
            credit =>
               actionbody(credit);
               state := 2
            debit =>
               actionbody(debit);
               state := 2
            close =>
               actionbody(close);
               state := 3
         endcase;
      3 =>
         case input of
               eof =>
                  skip
         endcase
   endcase
end account_dfa_inverted.
```

This text can be obtained by making three code level transformations to a DFA-based text for the ACCOUNT process

1) read statements are removed;
2) a parameter is introduced to provide input at the time of procedure invocation;
3) the outermost iteration is removed (effectively, to the calling procedure).

The advantages of this approach over standard inversion coding are as follows:

- The transformation is realised without `goto` statements, making it a viable approach to the implementation of JSD specifications in `goto`-less languages such as Smalltalk-80 and occam.
- While offering the advantages of Lewis's followmap-based approach (Section 4.2) with respect to implementation in Smalltalk, it will often be superior both in terms of space occupancy and execution time. The text will usually be shorter because it is generated from a DFA rather than a state-suboptimal NFA, and execution may be faster because branch conditions are implicit in the jump table underlying the `case` statement, rather than explicit in the serially evaluated guards of a multiway selection. There may also be a marginal gain over

standard inversion coding in execution efficiency. This derives from the saving of indirect jumps into the program text from a initial dispatcher.

- the code produced is more amenable to further transformation — this is demonstrated in Section 6.4;
- the code produced is more elegant and easy to understand, although this benefit is of little importance if transformational implementation products are regarded as 'object code'.

6.3 Context Filters

In order to preserve simplicity and to maintain a useful separation of concerns during modelling, the JSD developer assumes that inputs to a model process will be 'correct'. This correctness has both syntactic and semantic aspects. The condition for syntactic correctness is that strings of input messages will be valid life-histories as formally described by the structure diagram of the model. Semantic correctness, on the other hand, is more complex and is concerned with issues such as the validity of the values of action attributes, and the accuracy with which the model reflects the true world; it is consequently much more difficult (perhaps impossible) to formalise. This section is concerned with syntactic correctness.

Context filters (Jackson, 1983) have been briefly mentioned in Chapter 4. They are processes which receive input on behalf of model processes and ensure that it is syntactically correct before passing it on to the model. They achieve this by comparing each incoming action message with the current set of actions (sometimes refered to as the *context set*) that the model process is prepared to accept. Should a message arrive out of context, it is discarded, and the filter process begins a sequential search of the input stream for the next acceptable message. This method of input validation is similar to the symbol-skipping error recovery strategies utilised in the syntax analysis of programming languages (e.g., Backhouse, 1979).

The context set for each state of a process is directly represented in a DFA as the set of labels of the out-transitions from that state. Given a DFA D, it is straightforward to construct a GSM which behaves as a context filter D' by copying and updating the transition and effects table of D such that:

```
∀(s∈states(D), i∈αD)•
    -- shadow state transition and copy input to output
       (δD(s,i) ≠ nil ⇒  -- i.e., there is a valid transition on 'i'
          ((δD'(s,i) = δD(s,i)) ∧ (γD'(s,i) = [i])))
    ∧
```

```
-- stay in same state and copy ε to output .
   (δD(s,i) = nil ⇒ --i.e.,there is no valid transition on 'i'
      ((δD'(s,i) = s)) ∧ (γD'(s,i) = [ε])))
```

The result of applying this procedure to a DFA representing ACCOUNT is shown in Figure 6.1 (symbol skipping loops are shown in bold).

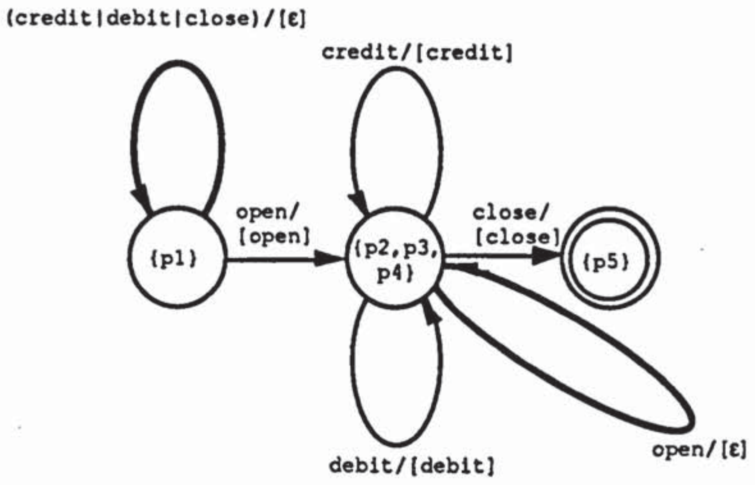


Figure 6.1. A GSM for context filtering the input to ACCOUNT.

For on-line applications, the effects table of the context filter may be elaborated so as to emit a diagnostic message on an error stream whenever a symbol-skipping transition is taken.

An Example

Imagine that ACCOUNT is to be implemented as an inverted DFA-based routine (see `account_ dfa_inverted` in Section 6.2) with its input filtered by a routine based on the GSM in Figure 6.1. The text for such a filter routine would have the following form:

```
procedure account_dfa_filter(input:msg_type; sv:sv_type; ok:
                                              boolean);
state renames sv.state;
   case state of
      1 =>
         case input of
            open =>
               account_dfa_inverted(input,sv);
               ok := true;
               state := 2
            otherwise =>
               ok := false
         endcase
      2 =>
         case input of
:
:
            etc...
```

```
:
:
end account_dfa_filter.
```

A scheduler for such a system might contain the following fragment for reading input for ACCOUNT from the external world and passing it on to the context filter:

```
procedure sys_sched;
:
:
-- set up a loop controlled by 'accepted'
accepted := false;
while not accepted do
(i)    get_next_action(action);
(ii)   get_sv(account_svdb, action.dest_id, sv);
(iii)  account_filter_dfa(input,sv,accepted);
       if not accepted then put(context_error_msg) endif
endwhile;
:
:
end sys_sched.
```

Salient features of this fragment are as follows:

(i) This line is assumed to call a user-interface routine to get the next desired `action`.

(ii) The state vector database for ACCOUNT is accessed, obtaining the state vector (`sv`) for the instance of ACCOUNT numbered by the destination identifier field of the `action` message.

(iii) The context filter is called with the `action`, the `sv` and the `accepted` flag. According to the value of `accepted` when the procedure returns, the scheduler will know whether the message was successfully passed on to the model, or whether it will need to request another `action` from the user.

In Section 6.5, a transformational technique is introduced which can be used to combine communicating routines such as `account_dfa_inverted` and `account_dfa_filter` into a single procedure.

6.4 Process Dismemberment

When implementing a JSD specification it is not always desirable to keep the entire text of a process together. The long-running processes obtained at the specification stage may have a conceptual life-time of several years and the text of such a process may include code which will be executed during only a short part of that life-time. In environments where resources are at a premium (for example in an embedded system with limited main memory, or a busy transaction processing environment) a transformational technique called *dismemberment* can be used to optimise code size and functional localisation (Jackson, 1983; Cameron, 1988). Using dismemberment, it is possible to implement a process as a number of separate pieces of program text which can be loaded and executed separately.

In most operating environments it will be unacceptable to allocate storage and other resources (for example, terminals or permission to access databases) to a process which will hold them for a long period without using them. In such circumstances it may be appropriate to dismember the process into a series of modules each of which can be allocated resources, scheduled and executed separately. These dismemberments can be made *by-state* or *by-input*, deriving modules to perform batch- and transaction-orientated processing respectively. For example, by collecting all the code of a process which would be executed on Wednesdays, say, a batch-orientated module is obtained. On the other hand, collecting all the code which is executed on receipt of a particular message, say a `debit` message, results in a transaction-orientated module. The distinction between batch and transaction orientations should not be taken too far — sometimes a dismembered module will deal with the processing associated with several states and several inputs. Furthermore, it should be noted that dismembered modules need not be disjoint.

Consider the library BOOK example once again. The business of running an imaginary library might be considered to have two facets: lending and stock control. Lending requires on-line support to allow the librarians at the loans counter to track the books as they are borrowed by members. Stock control activities, including the acquisition, classification and disposal of books, is attended to on Wednesday afternoons, when the library closes to the public. Stock control updates are to be made in batch mode on Wednesday night.

Figure 6.2 illustrates a possible implementation topology for the system. The vertical stripe on the system scheduler illustrates that it is an implementation routine with no analogue in the specification. The single bisected lines linking the pair of routines in each module indicate a call to a dismembered routine.

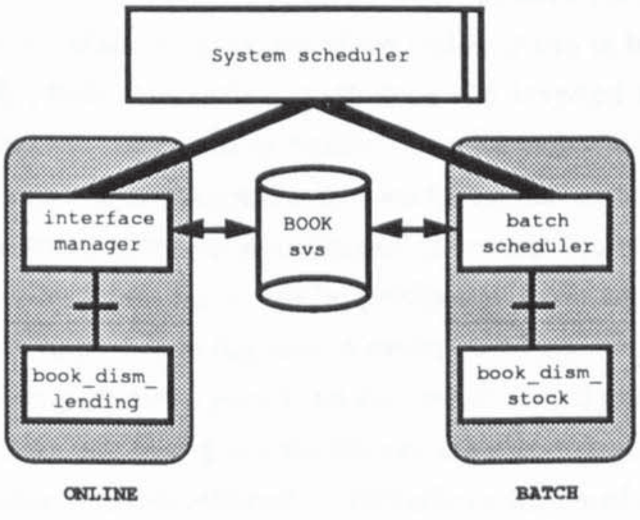


Figure 6.2. An implementation topology for part of a Library system.

Suppose that the system is to be implemented in two main modules — one online and one batch — called by an overall system scheduler (perhaps a job control script, or even a manual routine followed by the human system operator), together with a database of state vectors. As the counter librarians require a sophisticated user interface which will require a large amount of code to implement, the developer has decided to dismember the BOOK process so that only the code associated with loan activity is included in the on-line module. Stock control updates are to be made in batch by a dedicated module derived from the BOOK process specification by dismemberment. Below is shown the text of the on-line portion of the BOOK process.

```
procedure book_dism_lending(input: msg_type; sv: sv_type);
-- this just handles actions 'lend renew return'

state renames sv.state;
case state of
   in_library =>
      case input of
         lend =>
            actionbody(lend);
            state := on_loan
      endcase;
   on_loan =>
      case input of
         return =>
            actionbody(return)
            state := in_library
         renew =>
            actionbody(renew);
            state := on_loan
      endcase
endcase
end book_dism_lending.
```

Jackson (1983) illustrates the use of dismemberment for dealing with cases where an inverted subroutine needs to be called by two or more other routines running in an environment where subroutines cannot be shared. This can occur, for example, where the calling routines run as separate tasks, or one of the callers is run in batch and the other is run on-line. In such cases, it is necessary to copy the inverted routine so that it is separately available to each caller, and to ensure the synchronisation of the execution of each copy via shared access to the process's state-vector (which includes the text pointer). Provision must also be made to ensure mutually exclusive state vector access. Often, for a particular copy of a subroutine, there will be portions of code which are unreachable because they can be activated only in response to messages which will never be sent by that routine's caller. It is then possible to pare down the routine so that only the reachable code portions remain. The resulting saving in code size can often be significant.

The implementation freedom afforded by JSD allows the use of an existing operating system or a transaction monitor such as CICS to schedule dismembered modules. The executions of these modules are kept in their correct temporal relationship by common access to the process's state vector. A CICS-like system might allow a developer to implement a process as a set of small dismembered components and provide not only for appropriate resource allocation but also for automatic transfer of control and resources among them (Cameron, 1989a).

The question of appropriate use of dismemberment is not an easy one. Cameron (1989a) cautions against indiscriminately applying dismemberment to reduce module size. In many virtual memory environments, a large module is less likely to be swapped out than a smaller one (Peterson & Silberschatz, 1985). It may be preferable to therefore leave a text in a suboptimally large form, to ensure that it is readily available for execution. The advantage of the JSD approach, when supported by suitable software tools, is that various implementation options can be quickly generated and then tested in the actual environment.

Automation of Process Dismemberment

Given a DFA-based representation of a process, dismemberment is straightforward due to the explicitness of the relationships among states, inputs and transitions. Below is an algorithm which generates dismembered texts from a DFA of the type generated by subset construction:

```
procedure dismember
              (states:set of state; inputs:set of input;
   order:{s,i});
   if order = s
      then outerloop := states; innerloop := inputs
   elsif order = i
      then outerloop := inputs; innerloop := states
   endif;
   print('case' , outerloop , 'of');
   for o in outerloop do
      print(o , '=>');
      print('case' , innerloop , 'of');
      for i in innerloop do
         print(i ,'=>');
         print(γ(i,s) , ';
               state := ' , δ(i,s))
      endfor
   endfor
end dismember.
```

The algorithm provides a choice of input-major or state-major evaluation order, and constructs a program to handle only those states and inputs listed. State and input based dismemberments are available according to the chosen values of parameters. For example, to achieve dismemberment of a DFA D, the general form of procedure call would be as follows:

```
   dismember(S,A,order) .
```

Where the dismemberment is required by state, then `S = {s}` where `s` is the state the module is required for, $A = \alpha D$ and `order` $\in$ `{i,s}`. Note that it is probably better than `order` = `s`, so that the outer case statement tests the state once only. For dismemberment by input , `S = states(D)`, `A = {a}` (where `a` is the desired input), and order $\in$ `{i,s}`. Here, it is probably better than order = `i`, by similar reasoning to that above. Dismemberment by state & input would be produced by

```
        dismember({s},{a},order)
```

Here, `s` and `a` are as above, and `order` is not critical. Clearly, any other permutation of states and inputs is also obtainable. In the absence of an obvious choice for its value, `order` could be determined by the following heuristic:

```
      (card(S) < card(A) ⇒ order = i)    ∧
      (card(S) > card(A) ⇒ order = s)    ∧
      (card(S) = card(A) ⇒ order = (i ∨ s))
```

Chapter 7 describes a Smalltalk-80 implementation of this approach to dismemberment.

6.4 Network Strategies

Once the capability to dismember processes automatically has been achieved, it becomes realistic to consider implementation strategies which make greater use of the dismemberment transformation than would be practical with a hand-coded approach. In particular, one can think in terms of dismembering whole networks, that is, dismembering all the processes in a network and recombining the segments obtained into new modules. In this way one can alter the distribution of the processing in the network so that code is bundled into groups which see the processing of messages through from their input to the system to their ultimate effects (as updates to the state of the system and as outputs).

Consider the simple network in Figure 6.3.

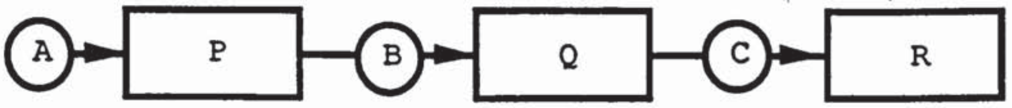


Figure 6.3. A simple datastream network.

Assume αA = `{w1,x1,y1,z1}`, αB = `{w2,x2,y2,z2}`, αC = `{w3,x3,y3,z3}`. Possible control flow patterns exhibited by an inverted implementation of this network is illustrated in Figure 6.5. Each message in αA causes a segment of the inverted routine to be called followed by a segment of its subordinate. So for example, the receipt by routine `P` of a `w1` message will lead to the invocation of routine Q with a w2 message and ultimately routine R with a w3 message. However, the code executed will just be the concatenation of the components of each routine which deal with w1, w2 and w3 respectively, and none of the x, y and z portions. The shadings in Figure 6.4 have been chosen to illustrate this visually.

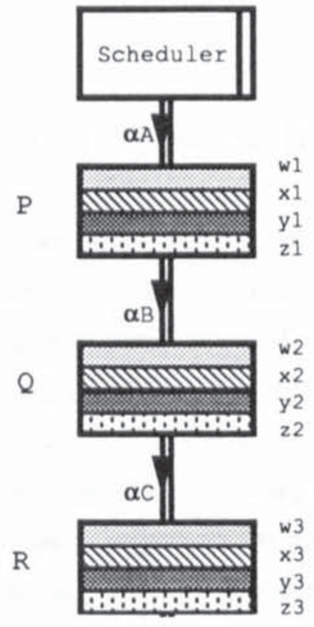


Figure 6.4. Execution patterns in a hierarchy of inverted procedures.

On the other hand, the coding obtained from a TDFD (see Chapter 2) design has the pattern illustrated in Figure 6.5. Each message is processed by a dedicated routine and control-flow semantics are much more straightforward. Many implementation environments are optimised for this configuration.

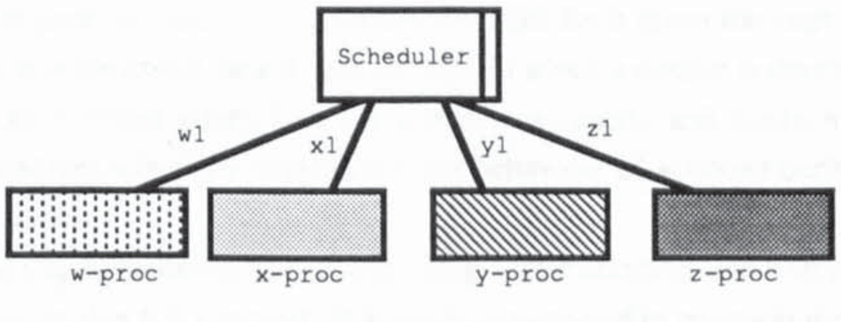


Figure 6.5. Execution patterns in a hierarchy of conventional procedures.

While the kind of topology illustrated in Figure 6.5 may represent a desirable implementation structure, it was argued in Chapter 2 that such a hierarchical structure is unlikely to provide a tractable abstract description of a large system. In particular, it can be very difficult to formulate an appropriate hierarchical structure from an analysis of the real world, and even assuming a useful hierarchy can be discovered, it is then likely to be difficult to change to accommodate drifting requirements. The network dismemberment strategy promises to allow the software developer to benefit from the operational nature of JSD (with the attendant advantages discussed in Chapter 2 — modelling of real world parallelism, executability, transformability etc), while being able to derive transformational implementations according to a 'conventional' scheme which may be more compatible with real-world implementation environments.

Towards an Implementation of Network Dismemberment

There will sometimes be situations where messages are passed unchanged through a pipeline of processes. This is the case when data is validated by a context filter on behalf of a model process, and in ensuring synchronisation of various roles of an entity (see Section 3.2). Implementing such a pipeline by the usual inversion route will necessitate the introduction of a procedure call to replace each of the write statements which provide input to the datastreams in the pipeline. Between each call, very little processing will generally be performed. Network dismemberment improves the ratio of substantive processing statements to procedure calls by collecting the processing statements into a single executable sequence called a *transaction module.*

Transaction modules can be generated by the composition of the appropriate dismembered components of each process which contributes to the processing of a particular action message. The resulting text is run once for each transaction. A feature of this composition is the ability to avoid some of the procedure calls introduced when inversion is used to implement write statements. This is done by *unfolding* each amenable procedure call (Darlington, 1982), i.e., replacing the call with the code which would be executed by the callee (see Section 2.3). Using the dismemberment facility developed in the thesis, is it possible precisely to identify this code for a given message type. Use of this facility is now illustrated for a simple example in which a module is developed from the specification of ACCOUNT which filters and processes `credit` and `debit` messages only (i.e., those messages which are important to the behaviour of ACCOUNT during the greater part of its life-time).

The first step is to convert the structure diagram for ACCOUNT into a DFA. From here, as explained in Section 6.3, a second DFA can be constructed to represent the context filter (`account_dfa_filter`, Section 6.3). If the DFA-based ACCOUNT is inverted with respect to its input (see `account_dfa_inverted`, Section 6.2), then the context filter will need to write to it by means of a procedure call. The relevant portion of text of the context filter is obtained by applying the following transformation:

```
    dismember(all, {debit,credit}, i)
```

This yields the following code fragment:

```
case input of
      debit =>
         account_dfa_inverted(input,sv);
         ok := true;
         state := 2;
      credit =>
         account_dfa_inverted(input,sv);
         ok := true;
         state := 2;
      otherwise =>
         ok := false;
endcase;
```

The calls to `account_dfa_inverted` can be unfolded by replacing them with the results of the following two dismemberments of its text respectively:

```
dismember(all,{debit},i)
```

yielding

```
sv.balance := sv.balance + input.amount;
```

and

```
dismember(all,{credit},i)
```

yielding

```
sv.balance := sv.balance + input.amount;    .
```

In general when unfolding a procedure call in this way, the names of the actual parameters used in a calling routine are unlikely to be the same as the names of the formal parameters of the called subordinate routine. For example, if the message parameter of `account_dfa_inverted` was called `msg` rather than `input`, then it would be necessary to introduce assignment statements to copy the value of `input` in the context filter to the corresponding variable `msg` in the text derived from `account_dfa_inverted`. This would result in the following text:

```
case input of
   debit    =>
         msg := input;
         sv.balance := sv.balance - msg.amount;
         ok := true;
         state := 2
   credit   =>
         msg := input;
         sv.balance := sv.balance + msg.amount;
         ok := true;
         state := 2
   other    =>
         ok := false;
endcase;
```

Although two procedure calls have been saved by applying the unfold transformation, two copy statements have been needed in their place. Aho *et al.* (1985) describe a technique called *copy propagation* which systematically renames the variable appearing on the left hand side of a copy statement with the name of the variable appearing on the right hand side from that point in the text onwards. This would yield the following text if applied to the above example:

```
case input of
   debit     =>
         input := input;
         sv.balance := sv.balance - input.amount;
         sv.state := 2
   credit    =>
         input:= input;
         sv.balance := sv.balance + input.amount;
         sv.state := 2
   other     =>
         ok := false
endcase;
```

The two statements `input := input` are redundant and can be removed using an optimisation called *dead code* elimination (Aho *et al.*, 1986).

Further investigation of network dismemberment is outside the scope of this thesis. A simple example has been chosen to illustrate how such an approach can improve the code-localisation of modules and save procedure calls. So far, it has been envisaged that the user would select the individual dismemberments to be made in implementing a network strategy. Some suggestions for an approach to the automation of this task are made in Chapter 8.

6.7 Concluding Remarks

Although structure diagrams may be a more convenient notation for capturing regular behaviour from an analyst's point of view (Zave, 1989, Renold, 1988b), finite automata provide a perspective which may afford significant advantages for the implementor. This chapter has illustrated four applications of DFA-based representations of JSD processes in transformational implementation:

- automatic generation of `goto`-less inverted texts;
- automatic generation of context filters;
- automation of process dismemberment;
- an approach to network dismemberment in which each calls to an inverted routine is replaced with the relevant dismembered portion of that routine using an unfolding technique.

The first three of these uses represent significant improvements to the NFA-based work of Lewis (1991) and provide the prerequisites for the fourth, which appears not to have been described elsewhere†.

† As of Summer 1992.

Chapter 7
Smalltalk-80 Implementation of DFA-based Transformations

7.1 Overview

The algorithms described in preceding chapters have been incorporated into the first-phase version of the PRESTIGE JSD implementor's workbench (Bass, Boyle & Ratcliff, 1991; Ratcliff & Boyle, 1992). Like the bulk of the PRESTIGE system, the software introduced here is implemented in the Smalltalk-80 programming language and runs on a Macintosh platform (and, in principle, in any other environment which provides a Smalltalk virtual machine).

The structure of the chapter is as follows. To provide orientation, an overview of the PRESTIGE workbench is presented in the next section. Particular attention is given to the process-level facilities of the toolkit; code generation in both untransformed and inverted forms is described. The third section describes the facilities added to the PRESTIGE workbench to implement the automatic generation of DFAs from process structures. The fourth section describes the implementation of dismemberment for generating both self-contained modules and transaction handler subtexts.

The PRESTIGE Workbench

The PRESTIGE project is an ongoing venture intended to provide software support for JSD implementation (Bass *et al.*, 1991). Recall from Chapter 2 that JSD is an operational method and that implementation consists of the (automatable) transformation of an executable abstract specification. In keeping with the operational approach, a basic principle of the PRESTIGE philosophy is that the binding of the progressively transformed product to its intended implementation environment should occur as late as possible. To this end, a two-step implementation process model has been adopted. *Primary* transformation first produces an intermediate form expressed in a representation independent of the chosen target language. The intermediate form then undergoes *secondary* transformations by what may be loosely termed a 'code generator' to yield the desired implementation. A diagrammatic representation of the overall implementation process supported by PRESTIGE is given in Figure 7.1.

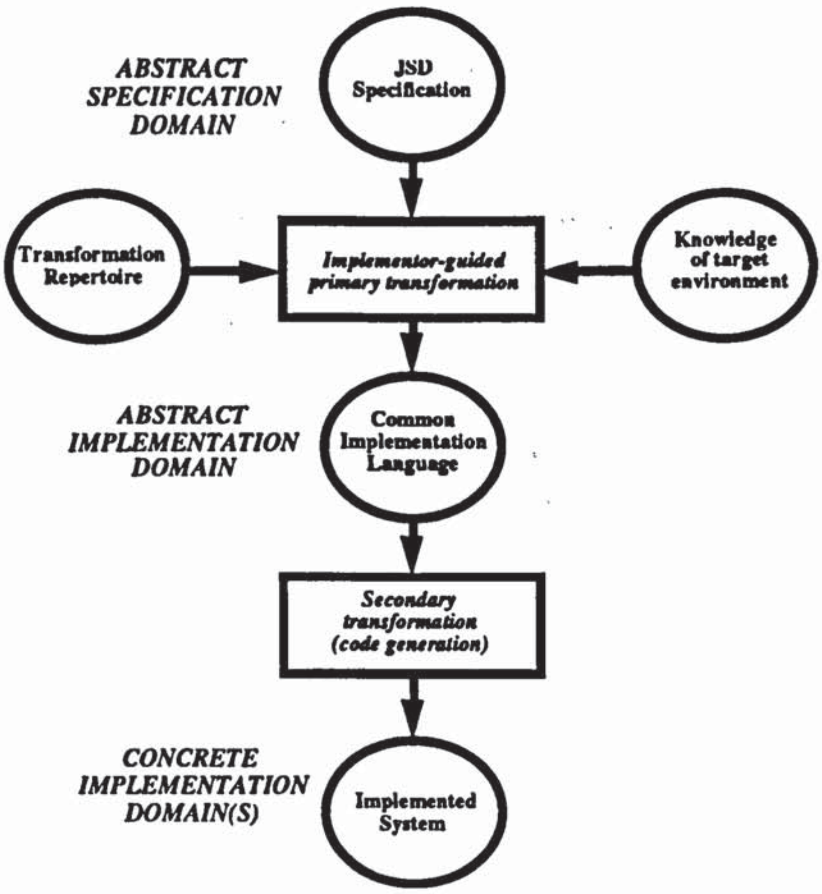

ABSTRACT SPECIFICATION DOMAIN
JSD Specification
Transformation Repertoire
Implementor-guided primary transformation
Knowledge of target environment
ABSTRACT IMPLEMENTATION DOMAIN
Common Implementation Language
Secondary transformation (code generation)
CONCRETE IMPLEMENTATION DOMAIN(S)
Implemented System

Figure 7.1. Basic implementation process model supported by PRESTIGE (after Bass *et al.*, 1991).

Bass *et al.* (1991) advance the following three arguments in support of this two-stage implementation model:

- There is a useful modularisation of 'transformational concerns' between aspects that are quite independent of the target language and those that are coupled to it.
- Toolkit generality, flexibility and portability are enhanced. In principle, all that is needed to customise PRESTIGE is to hook on to its primary component an appropriate secondary code generator.
- Reusability and maintainability are supported — that is, reuse and maintenance of *specification* components and intermediate products. For example, if an existing specification is subsequently targeted at a new environment, then it may be necessary only to transform the previously generated intermediate form using an appropriate code generator.

Commitment to the primary-secondary transformational model has necessitated a way of representing the products of the first transformational phase within the toolkit. This task has been fulfilled by the use of an intermediate form called the *Common Implementation Language* (CIL). CIL has twin design aims:

- to represent JSD-specific implementation structures (such as inverted routines and state vector databases) without commitment to a particular concrete implementation environment;
- to be easily translatable into a variety of actual procedural languages.

CIL is *not* a new programming language. It is not intended to be visible to the implementor, and is described as a 'language' only to facilitate easy explanation of its form and purpose.

To allow the developer to specify assigned operations in a way independent of a particular programming language, a Pascal-like design language called ESTEL is provided. The toolkit has facilities to allow the editing of ESTEL and can translate the language into a variety of compilable codes.

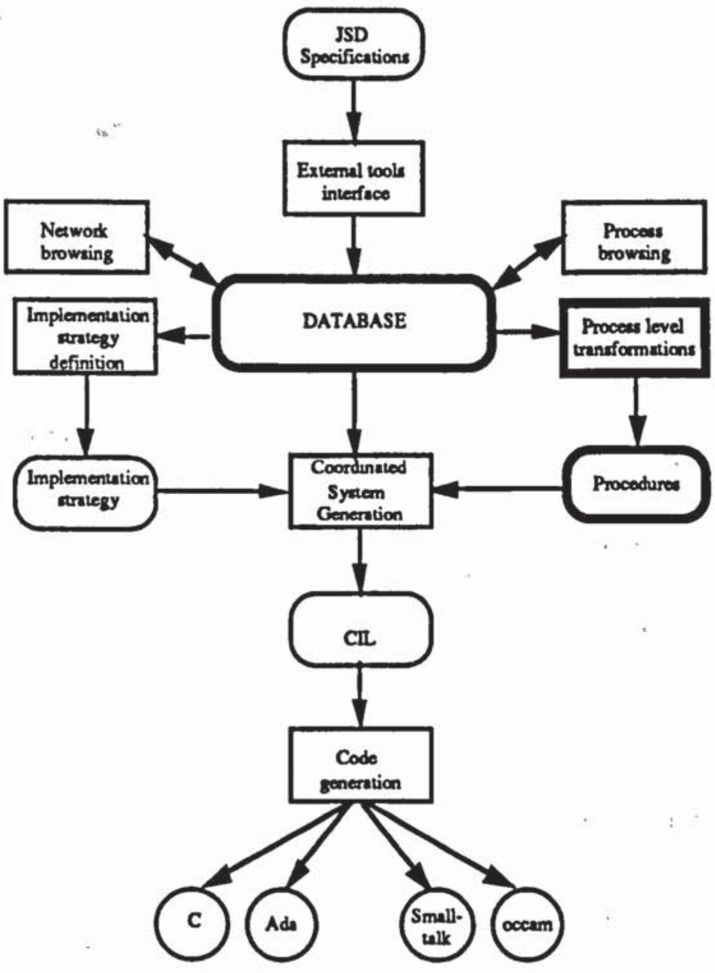


Figure 7.2. Functional overview of PRESTIGE.

Figure 7.2 provides a functional overview of the toolkit. Its major features include a repertoire of transformations; automatic generation of default system topologies according to the 'knitting needle' strategy of Cameron (1986); a graphical user-interface and interactive validation of the applicability of transformation decisions. The highlighted portions of the diagram illustrate the part of the toolkit's functionality with which the present work is concerned — that is, the process-level facilities of the toolkit.

7.2 Process-level Facilities

The PRESTIGE toolkit provides facilities for editing structure diagrams and generating code from these diagrams as either untransformed process texts or inverted subroutines. The code generation facilities are described by Bass *et al.* (1991) and Ratcliff & Boyle (1992). The implementation of the structure diagram editor is described by Lewis (1991).

Figure 7.3 shows the hierarchical organisation of the Smalltalk classes involved in the implementation of the process-level facilities offered by PRESTIGE.

```
Object "Parent of all classes in a Smalltalk system"

   AbstractLanguage "Perform code generation from <ESTELConstructs>"
      Ada
      StructureText

   Collection
         Dictionary
            OperationTable "Instances store allocated operations"

   DfaOps "Instances represent JSD processes as finite automata"

   ESTELConstruct "Parses assigned operations"

   Model
      Browser
         Structure "Instances represent structure diagrams"
            StructureWithComms
               StructureWithFollowset "+ followset computation"
                  JSDProcessBrowser "+ user interface"
                     ImplProcess "+ code gen. and inversion"

   Node "Instances represent nodes making up <Structures>"
      CodeNode " + ability to be allocated with operations"
         ItrNode
         SelNode
         SeqNode
```

Figure 7.3. Hierarchical organisation of the Smalltalk classes implementing process-level facilities.

The Tree-walking Code Generation Algorithm

Basically, an exhaustive preorder walk of the structure is performed, with control structure headers and footers being generated during the traversal as appropriate. Figure 7.4 provides a key to the notation used to explain this tree-walking approach. In this diagram, the dashed lines represent control flow into and out of `Node`s, and the textual annotations describe in abstract terms the work performed by the algorithm at various points.

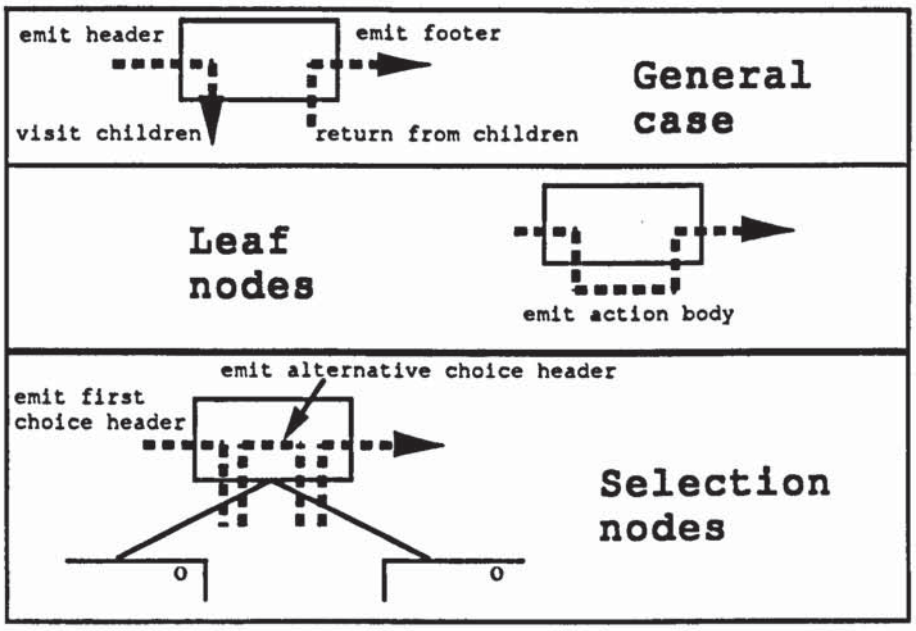


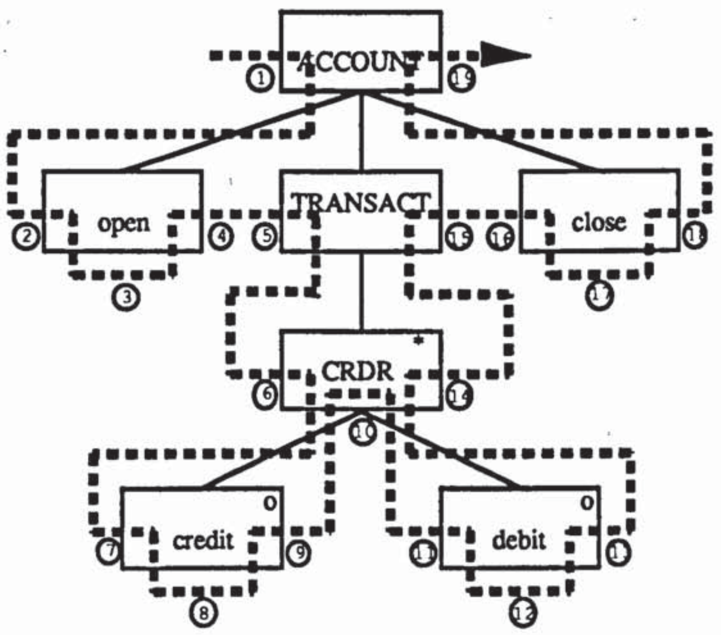
Figure 7.4. Control flow through nodes during tree walking code generation

Textual representations of process structures are generated by the method `code:inverted:` to instances of `ImplProcess`. A graphical trace of the operation of this method on the ACCOUNT process is shown in Figure 7.5. The circled numbers relate points in the traversal to emitted lines of code. A few lines require particular comment:

1&3) read statements are emitted at the beginning of the text and then immediately after the generation of each action body;

5) the condition on an iterated subtree T is that the current `input` message is not a member of `FOLLOW(T)`.

6&10) The conditions on each branch of a selection are that the current `input` message is equal to the name of the root of the respective branch.



```
①  ACCOUNT seq read(input);
②   open seq
③       action_body(open); read(input);
④   open end;
⑤   transact itr while input <> close
⑥     crdr sel input = credit
⑦       credit seq
⑧         actionbody(credit); read(input);
⑨       credit end;
⑩     crdr alt input = credit
⑪       debit seq
⑫         actionbody(debit); read(input);
⑬       debit end;
⑭     crdr end;
⑮   transact end;
⑯   close seq;
⑰     actionbody(close);
⑱   close end;
⑲  ACCOUNT end.
```

Figure 7.5. Tree-walking code generation from the ACCOUNT structure.

Generation of Action Bodies

Each leaf node is asked to generate its associated action body by sending the message

```
generate: pLang  indent: indent  on: outStream  transf:
invFlag  opTable: exOpDict
   "Generate a sequence of containing your assigned operations in
   <pLang>, indented <indent> spaces.  Use your list of assigned
```

```
        operations to get the code fragments from <opTable>.  Invert
        according to the boolean <invFlag>.Emit code on <outStream>".
```

This method, implemented in class `Node`, achieves its purpose by accessing a table of executable operations using a list of keys describing the node's actionbody. This table, called the `exOpDict`, is an instance variable of the parent process structure, and is itself an instance of class `OperationTable`. Figure 7.6 illustrates a possible instantiation of `OperationTable` for the ACCOUNT process.

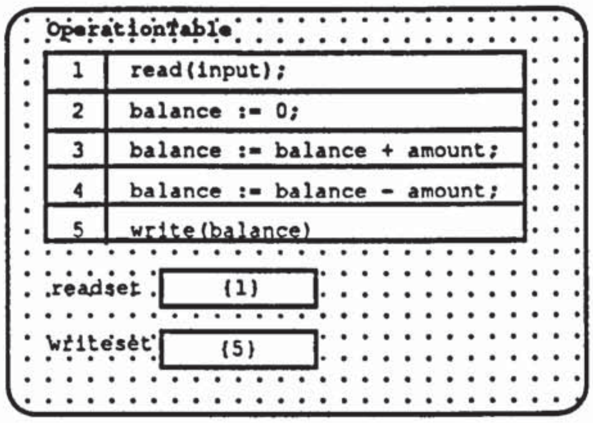


Figure 7.6. An instantiation of `OperationTable` for the ACCOUNT process.

Class `OperationTable` is implemented as a subclass of `Dictionary` (Smalltalk's standard associative array class), and assuming that no transformations are to be performed, they behave as arrays of program fragments. These program fragments are stored as ESTEL syntax trees and can be used to generate code in a variety of programming languages including Ada and Smalltalk.

Implementation of Inversion

`OperationTables` can transform ESTEL read and write operations into suspend and resume instructions for the purpose of realising inversion. The generation of inverted texts proceeds essentially in the same tree-walking manner as above, except that the `OperationTable` is asked to return read and write operations in their transformed form. The method used to access these operations is `at:transf:`, shown below:

```
at:  index  transf:  trFlag
   "If  trFlag  is  false,  use  inherited  functionality  from
<Dictionary>"
       trFlag ifFalse: [↑super at: index].

   "Similarly, if the operation is not a read or write,
   no transformation necessary"
      ((readSet includes: index)
         or: [writeSet includes: index])
         ifFalse:
            [↑ super at: index].
   "Consult record of implementation topology for appropriate
inversion  mode, & if absent make it read inversion by default"
      transformation ← ImplTopology at: (super at: index)
dataStream
        ifAbsent: [transformation ←#RINV].

   "return appropriately transformed i/o operation"
     ↑self at: index withTransformation: transformation
```

The exact substitutions to be performed depend on the implementation topology chosen by the developer (Jackson, 1983).

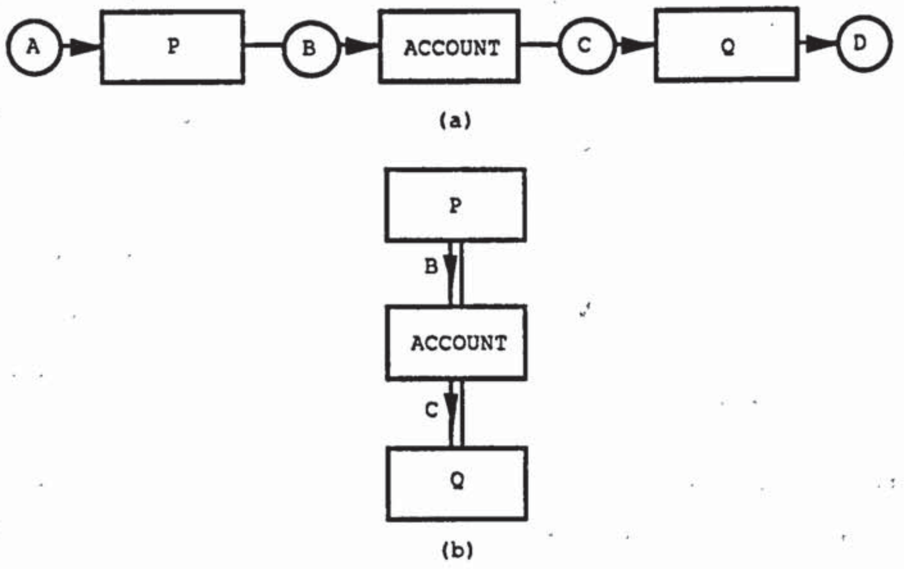


Figure 7.7. (a) A network incorporating the ACCOUNT process.
(b) A possible implementation procedure hierarchy.

If ACCOUNT is embedded in the network shown in Figure 7.7(a), and is to be implemented as part of the topology shown in Figure 7.7(b), then the messages `at: 1 transf: true` and `at: 5 transf: true` will yield the following transformed program fragments respectively:

```
tp := "next label"; return;
   -- next label = (number of calls to at:transf: this walk) - 1

call(Q, balance);
```

Appendix I provides a Smalltalk listing of the implementation of `OperationTable`.

7.3 Implementation of DFAs

DFAs are represented in Smalltalk by the class `DfaOps`. Instances of this class have the following variables : `name`, `start`, `tt`, `et`, `alphabet` and `finals`. These have the meanings described in Chapter 4. A full listing of the implementation of `DfaOps` is given in Appendix I.

Instances of `DfaOps` have protocol to support a variety of code generation schemes, together with on-the-fly regular language recognition. Of the instance variables of `DfaOps`, two are particularly important: `tt`, which represents the transition function of the `DfaOps`, and `et` which represents the output function. Both tables are implemented as instances of subclasses of `TransitionTable`.

`tt` understands two key messages concerned with state transitions:

```
from: aState1 to: aState2 on: action
   "Build a transition from <aState1> to <aState2> labelled
   <action>"

delta: aState on: anAction
   "Answer the destination state reached by accepting <anAction>
   in <aState>"
```

These are used to construct and animate DFAs.

Regular language recognition is provided by the method

```
recognise: aList
   "Answer whether or not you recognise <aStream>" .
```

This method, if passed a list of action names, will decide whether or not the list constitutes a valid life-history of the process by simulating the behaviour of the automaton. For example, if `bookDFA` is an instance of `DfaOps` derived from the `BOOK` process then the expression

```
bookDFA recognise:
   #(acquire classify loan return load renew renew return sell)
```

would evaluate to `true`. In Section 8.4, suggestions are made for further work in this area with the aim of providing a rapid prototyping facility for the JSD modeller.

Subset Construction

Class `ImplProcess` has been provided with protocol to construct DFAs using the subset construction method. In doing so, it calls upon the following auxiliary methods:

```
alphabet
firstPos
followPos: aPosition
nameOf: aPosition
```

These have the semantics previously described in Chapters 4 and 5, and their implementations are listed in Appendix I, together with the various subset construction algorithms. The methods `firstPos` and `followPos` were modified from code implemented by Lewis (1991) for the calculation of FIRST and FOLLOW functions (see Chapter 3).

Below is presented the implementation of the algorithm for the construction of a transition table for a DFA from an ImplProcess.

```
dfa
   "transform into a dfa using a subset construction algorithm
   based on that of Aho et al. 1985"

   | dStates unMarked t inPositions u tt |
   tt ← TransitionTable new.
   dStates ← MarkedSet new.

   "Dstates := {firstpos(root)};"
      dStates add: self firstPos.

   "while there is an unmarked state T ∈ Dstates do"
      [(unMarked ← dStates select: [:s | s marked not]) isEmpty]
         whileFalse:
            [t ← unMarked asOrderedCollection first.

            "mark(T);"
               t mark: true.

            "for each input ∈ αM do"
               self alphabet do:
                  [:input |

                  "inPositions := {p ∈ T | nameof(p) = input};"
                     inPositions ← t select:[:p |(self nameOf: p)
                        = input].

                  inPositions isEmpty ifFalse:
                     ["U := ∪ {fset ∈ PN1 | q ∈ inPositions ∧
                                             fset = followpos(q)}"
                        u ← self distrUnion:
                              (inPositions collect: [ :pos |
                                    self followPos: pos])].

                     "if U ≠ {} and U ∉  Dstates
                     then Dstates := Dstates ∪{ U } endif;"
                        u isEmpty | (dStates hasMember: u)
                              ifFalse: [dStates add: u].

                     "Dtran[T, input] := U;"
                        tt from: t to: u on: input]].

   ↑DfaOps
      name: self name start: 's1' transitions: tt
      effects: nil alphabet:self alphabet finals:(Set new add: #s)
```

The text has been commented with fragments from the pseudocode used to present the design of this algorithm in Chapter 4 (in italics). The implementation of the general algorithm for subset construction, allowing for operations and recognition difficulties, is presented in Appendix I.

7.4 Dismemberment

Class `DfaOps` provides the following instance protocol for code generation:

```
hardcode
   "Generate a hardcoded version of the entire DFA"

dismemberedStates: states inputs: inputs
   "Generate a hardcoded version of the portion of the DFA which
   includes <states> and has transitions labelled by <inputs>"

invertedDismemberedStates: states inputs: inputs
   "As above, except include appropriate code to allow the text to
be    inserted into a hierarchy of inverted procedures"
```

This section provides some examples of the application of these methods to JSD specifications. The implementations are given in Appendix I.

The BOOK Process Dismembered by State

This is a straightforward application of dismemberment. Imagine that it has been decided to implement modules to deal with three phases of the life of a book. The first is at the beginning of its life as it is acquired, classified and made available to lenders, the second is the active part of the life-history, during which time it is being lent, renewed and returned, and the third is the part of its life where it is being disposed of. These three modules are quite disjoint from a scheduling perspective — there is no need to load the code associated with `classify`ing a book when handling a `renew` message, for example. Constructing the DFA yields a transition table which is pretty-printed by Smalltalk as follows:

```
TransitionTable
   (s78->
      Dictionary
         (swap->s
         sell->s )
   s1->
      Dictionary
         (acquire->s2 )
   s36->
      Dictionary
         (outcirc->s78
         lend->s45 )
   s2->
      Dictionary
         (classify->s36 )
   s45->
      Dictionary
         (return->s36
         renew->s45 ))
```

A TransitionTable can be understood as a map from each state to a Dictionary which records transitions from that state on each acceptable input. The tool-generated name of each state is made up of an initial s suffixed by a list of positions of the original tree which can be matched from the state. Examination of the states built by the construction algorithm shows which ones need to be included in each of the modules. These can then be generated with the following calls:

```
newBook ←
   bookStructure
      invertedDismemberedStates: #(s1 s2)
      inputs: inputs.
useBook ←
   bookStructure
      invertedDismemberedStates: #(s36 s45)
      inputs: inputs.
disposeBook ←
   bookStructure
      invertedDismemberedStates: #(s78 s)
      inputs: inputs.
```

The code generated for book_fsm_use is shown below. To make the structural features of the example clear, no assigned operations have been inserted into this text.

```
procedure book_fsm_use(input);
case state of
s36 =>
   case input of
      outcirc =>
         -- ops
         state := s78;
      lend =>
         -- ops
         state := s45
   endcase;
s45 =>
   case input of
      return =>
         -- ops
         state := s36
   endcase;
   case input of
      renew =>
         -- ops
         state := s45
   endcase;
endcase;
end book_fsm_use;
```

One possible scheduling strategy to exploit the dismemberment of BOOK described here would be to run the modules book_fsm_new and book_fsm_dispose in batch mode (perhaps weekly), and run the module book_fsm_use on-line. See Appendix II for listings of the batch modules.

The Regular Expression `(a b| a c)* da` Dismembered by Input

This example is interesting because it contains both a recognition difficulty and multiple synonymous leaves (the two cases for which Lewis (1991) was unable to cater). For illustrative purposes, the simple operation map shown in Table 7.1 will be assumed.

position	action body
p1	[1]
p2	[2]
p3	[3]
p4	[4]
p5	[5]
p6	[6]

Table 7.1. An `operation_map` for $\mathrm{ACCOUNT}_{ops}$.

If a process structure representing this expression is stored in a variable `regex`, then a complete text in its DFA-based form can be obtained by evaluating the following statement:

```
(regex dfaOpsRecogn) dismemberedStates: #all inputs: #all
```

The code produced is presented below:

```
case state of
s135 =>
   case input of
      d =>
         [5];
         state := s6;
      a =>
         [save,1];
         state := s24;
   endcase;
s24 =>
   case input of
      b =>
         [2];
         state := s135;
      c =>
         [restore,3,4]
         state := s135
   endcase;
s6 =>
   case input of
         a =>
            [6];
            state := s;
   endcase;
endcase;'
```

As this procedure contains no `goto` statements, it can be directly transcribed into languages such as Smalltalk and occam, so providing a way of implementing backtracking in these languages. Two examples of dismembered fragments of this procedure are now described.

They are respectively, (i) which deals with all a inputs, and so caters for the positing of a particular branch and also illustrates the handling of synonymous leaves; (ii) one which deals with c inputs and so caters for the admission of an incorrect posit. The call

```
(regex dfaOpsRecogn)
   dismemberedStates: #(s135 s24  s6) inputs: #(a)
```

leads to the generation of the following text:

```
case state of
   s135 =>
      case input of
         a =>
            [save,1];
            state := s24;
      endcase;
   s6 =>
      case input of
         a => [6];
         state := s
      endcase;
endcase;
```

The call

```
(regex dfaOpsRecogn)
   dismemberedStates: #(s135 s24 s6) inputs: #(c)
```

produces

```
case state of
   s24 =>
      case input of
         c =>
            [restore,3,4];
            state := s135;
   endcase;
endcase.
```

The ability to dismember processes which incorporate backtracking behaviour is a major advantage of the approach taken in the thesis.

7.6 Concluding Remarks

This chapter has discussed an implementation of the ideas developed in the thesis, so demonstrating their feasibility. These ideas have been incorporated into a version of the PRESTIGE JSD Implementor's toolkit and have been coded in the Smalltalk-80 programming language. By way of introduction to the new transformations, salient features of the PRESTIGE environment have been discussed, including its existing code generation and inversion facilities. A guide to the subset-construction based implementation of the new dismemberment facilities has been provided, along with the results of their execution on some examples. These examples illustrated:

- production of batch and on-line modules by state dismemberment;
- production of an input-driven module from a structure containing both a recognition difficulty and multiple synonymous leaves;
- production of transaction components for use in network dismemberment implementation schemes.

Further work is ongoing to integrate these capabilities with other PRESTIGE-related work and these are discussed in the next chapter, together with conclusions and other suggestions for further work.

Chapter 8
Conclusions

8.1 Re-representation as a Transformational Strategy

In a short paper, Peterson (1992) discusses a particularly powerful strategy for the solution of S-type (fixed formalisable) problems (Lehman, 1980) such as those traditionally discussed in the AI literature, for example the 'eight queens' (Balzer, 1982) and 'missionaries and cannibals' problems (Rich, 1990). This strategy, known as *re-representation*, involves changing the representation of a problem prior to attempting to solve it. Figure 8.1 shows a model of the re-representation approach to problem-solving essentially as given by Peterson, instantiated to show the links with the present work.

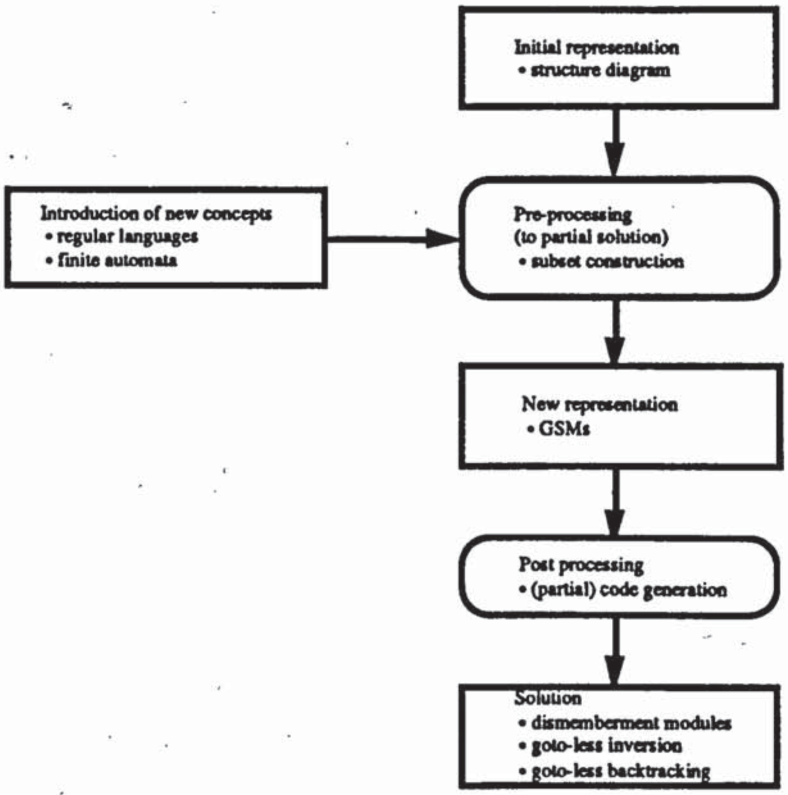


Figure 8.1. The approach taken in the thesis regarded as a re-representation strategy.

This work has established a generic re-representation approach to the implementation of JSD processes. It is generic in the sense that it solves the problem of transforming the *class* of syntactically well-formed process structures into a variety of implementations via their re-representation as DFAs. This approach is distinct from the one taken by Cameron (1989d) and mentioned in Chapter 1, precisely because it involves re-representation where Cameron's does not. Given the specific goal of implementing dismemberment, it is difficult to favour one approach over the other. It is felt, however, that the re-representational approach presented here is more general as it offers potential support a variety of other development activities (to be discussed in Section 8.4)

The rest of this chapter takes the following form. First, the contribution of the thesis is summarised. Secondly, weaknesses and limitations of scope are acknowledged. Finally, some suggestions are offered for further development.

8.2 Summary of Contribution

This work has shown how to transform JSD process specifications into finite automata and, based on this new representation, how to effect process dismemberment and `goto`-less inversion for syntactically well-formed Jackson trees. This appears to be the first description of automated dismemberment by this method†. The only other known description of work on the automation of dismemberment is given in a privately circulated paper (Cameron, 1990) using a different approach.

The present work owes a debt to the doctoral thesis of Lewis (1991) which provided the insight that (language-theoretic) follow-set representations of JSD processes could be used to control their execution. By more deeply studying the link between JSD and the applied formal language theory used by compiler writers, this work has been able to address the main limitations of Lewis's work. The improvements can be summarised as follows:

- Far fewer states are introduced in the present approach, because rather than identifying a unique state with *each* of a process's actions, the new approach recognises some such states to be equivalent.
- By explicitly constructing transition tables for processes, the opportunity to evaluate transitions by table-lookup or `case` statement has been provided. This affords significant efficiency gains over the sequential evaluation of multi-way selection conditions as employed by Lewis.
- Recognition difficulties can now be handled (this work appears to be the first description of an efficient `goto`-less single lookahead implementation of backtracking†).
- Multiple synonymous leaves can now be handled.

† As of Summer 1992

In addition to providing an implementation of the dismemberment transformation, and to improving upon the efficiency and generality of Lewis's approach, this research has also yielded the opportunity to support a new implementation strategy involving the dismemberment of whole networks. The application of network dismemberment to the generation of transaction modules dedicated to processing particular message types has been illustrated, and the foundations have been laid for further work in this area.

8.3 Limitations

Limitations are discussed in two parts: conceptual and practical.

Conceptual Limitations

The major conceptual limitation is that the approach only applies to well-formed Jackson structures as defined by Hughes (1979). Features which the definitive JSP/JSD literature mentions which are not therefore supported are:

- operations assigned to null nodes (e.g. Cameron, 1989);
- operations assigned to interior nodes of structure diagrams (e.g. Jackson, 1975);
- quits based on the values of action or entity attributes (e.g. Jackson, 1983).

The first two of these are not felt to be serious as their effects are achievable using well-formed trees (backtracking can be used to circumvent the null nodes problem, while 'redundant' sequence nodes can be used to allow the behaviours generally achieved by the use of interior-assigned operations). The third problem seems outside the scope of this approach. It is closely related to exception handling, which appears difficult to express in language-theoretic terms.

No consideration has been given to fixed merged input, in which a process alternately reads from two or more input streams, or to conversational constraints or controlled datastreams.

Limitations of the Practical Work

The objective of the practical work has been simply to demonstrate the validity of the concepts developed here. Therefore integration with the user-interface and real-world code generation facilities of the PRESTIGE system (Bass *et al.*, 1991) has been rudimentary. In particular, DFA datastructures are available for inspection only in a codified form, and the user is required to examine the datastructure directly to decide which states to include in a dismembered module. In addition, to simplify the design of the practical component of the work, code is generated in an *esperanto* language related to Jackson structure text (Jackson, 1983) rather than an actual compilable programming language.

8.4 Suggestions for Further Work

The current software provides a poor interface to the datastructures which represent DFAs. An obvious future development would therefore be a *DFA browser*. Such a software tool would allow the implementor to interact directly with a transition diagram to prescribe particular dismemberments of processes (the explicit representation of states and inputs seems to make the transition diagram a more convenient notation for considering potential dismemberments than the structure diagram). The JSD modeller also stands to benefit from the facility to generate automatically an alternative, more dynamic, view of an entity's behaviour. This aspect of the browser could be enhanced further by offering a prototyping facility based on the `recognise:` animation method described in Section 7.3. Using this facility, a modeller could confirm that a particular structure diagram did indeed admit a representative selection of life-histories, and could validate such a description with a client.

Jackson methods feature control abstractions which are not generally directly available in programming languages (particularly inversion and backtracking), and the Jackson implementor has therefore been required to simulate these abstractions using `goto`-statements. This has meant that use of the methods in object-oriented and process-oriented languages has been limited. Lewis (1991) shows how inversion can be achieved in Smalltalk using a followset-based approach, and this thesis has built on his work to improve the efficiency of the `goto`-less inversion transformation and also facilitate the implementation of backtracking. There is obvious potential for integrating this work with Lewis's to broaden the applicability of his tool. This could be undertaken within the framework of the PRESTIGE system. The potential to implement occam systems using the same transformations could also be explored.

Network dismemberment has been introduced in this thesis and there exists much scope for further work to consider both the appropriate application and implementation of the technique. Network dismemberment appears well suited to transaction processing environments for which it could be used to generate transaction modules dedicated to processing particular types of incoming messages. The issues requiring investigation include the management of shared access by concurrent transactions to state vectors, performance trade-offs with respect to other implementation strategies (based, for example, on inversion), and the automatic selection of dismembered process components for inclusion in transaction modules.

Zave (1985) notes that certain JSD-style networks can be transformed into single monolithic finite automata by the composition of the automaton describing each process. These automata can then be analysed for potential deadlock or ambiguity quite simply (deadlock risk exists when there exists a state from which there is no out-transition which can ultimately lead to an accepting state, and ambiguity exists when there is a choice of transitions from a single state on a given input). Furthermore, such automata should be amenable to performance analysis if execution times are associated with the transitions (based on the number and type of executable operations to be performed). Production of such automata appears to be related to the automatic dismemberment of whole networks, and could be a fruitful avenue for further investigation, if only to circumscribe the limits of such an approach.

References

Agresti, W W. *New Paradigms for software development.* Washington D.C: IEEE Computer Society Press; 1986.

Aho, A V; Sethi, R; Ullman, J D. *Compilers, principles, techniques and tools.* Reading, Mass: Addison-Wesley; 1985.

Alexander, H; Jones, V. *Software Design and Prototyping using Me too.* London: Prentice-Hall.

Avison, D E. *Information Systems Development: A Data Base Approach.* Oxford: Blackwell Scientific Publications; 1987.

Avison, D E; Fitzgerald, G. *Information systems development: methodologies, techniques and tools.* Oxford: Blackwell Scientific; 1988.

Backhouse, R C. *Syntax of Programming Languages: theory and practice.* London: Prentice-Hall; 1979.

Balzer, R. *Transformational Implementation: An Example.* IEEE Transactions on Software Engineering; 1981; 7(1): 3-14.

Balzer, R; Goldman, N M. *Principles of Good Specification and their Implication for Specification Languages.* Gehani; McGettrick, eds. Software Specification Techniques. Reading, Mass.: Addison-Wesley; 1986: 25-39.

Balzer, R; Goldman, N M; Wile, D S. *Operational Specification as the Basis for Rapid Prototyping.* ACM SIGSOFT Software Engineering Notes; 1982; 7(5): 3-16.

Bass, A P; Boyle, M; Ratcliff, B. *PRESTIGE: a CASE workbench for the JSD implementor.* In: 13th International Conference on Software Engineering. Los Alamitos, CA: IEEE Computer Society Press; 1991: 198-207.

Bergland, G D. *A Guided Tour of Program Design Methodologies.* Computer Magazine; 1981; October: 18-37.

Boehm, B W. *A Spiral Model of Software Development and Enhancement.* Computer; 1988; 21(5): 61-72.

Boehm, B W. *Software Engineering.* IEEE Transactions on Computers; 1976; 25(12): 1226-1241.

Booch, G. *Object oriented design with applications.* Redwood City, CA: Benjamin/Cummings; 1991.

Borgers, M; Munro, M. *Producing Better Maintainable JSD Specifications by Grouping Common Aspects.* Software Maintenance: Research and Practice; 1990; 2: 61-80.

Bornat, R. *Understanding and writing compilers: a do-it-yourself guide.* London: Macmillan; 1979.

Boyle, J M; Muralidharan, M N. *Program Reusability through Program Transformation.* IEEE Transactions on Software Engineering; 1984; SE-10(5): 574-588.

Brooks, F P. *No Silver Bullet - Essence and Accidents of Software Engineering*. In: Kugler, H J. Information Processing 86. North-Holland: Elsevier Science Publishers B.V; 1986: 1069-1076.

Broy, M. *Formal treatment of concurrency and time*. McDermid, J, ed. Software Engineer's Reference Book: Butterworth-Heinmann; 1991.

Broy, M; Wirsing, M. *Methods of programming. Selected papers on the CIP-Project*. Berlin: Springer-Verlag; 1991.

Burstall, R M; Darlington, J. *A transformation system for the development of recursive programs*. Journal of the ACM; 1977; 24(1): 44-67.

Cameron, J R. *A Brief Description of Work in Progress on Dismemberment*. Privately Circulated; 1990.

Cameron, J R. *An Overview of JSD*. IEEE Transactions on Software Engineering; 1986; 12(2): 222-240.

Cameron, J R. Personal Communication; 1989.

Cameron, J R. *Prototyping core functionality using JSD*. IEE Colloquium on 'Requirements Capture and Specification for Critical Systems'. London: IEE; 1989.

Cameron, J R. *The modelling phase of JSD*. Information and Software Technology; 1988; 30(6): 373-383.

Cameron, J.R. *JSP and JSD: The Jackson approach to software development* (second edition). Washington: IEEE Computer Society Press; 1989.

Cameron, J.R. *Mapping JSD Network Specifications into Ada*. Ada User Supplement; 1987: 591-599.

Cheatham, T E. *Reusability Through Program Transformations*. IEEE Transactions on Software Engineering; 1984; SE-10(5): 589-594.

Chomsky, N. *Syntactic Structures*; The Hague; 1957

Darlington, J. *Program Transformation*. In: Darlington, J; Henderson, P; Turner D A, eds. Functional Programming and its Applications. Cambridge: Cambridge University Press; (1982).

Davis, R E. *Runnable Specification as a Design Tool*. In: Clark, K L; Tärnlund, S A. Logic Programming. London: Academic Press; 1982.

DeMarco, T. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall; 1979.

Dijkstra, E. *A Discipline of Programming*. Englewood Cliffs, N.J: Prentice-Hall; 1976.

Dromey, R G; Chorvat, T A. *Structure Clashes - An Alternative to Program Inversion*. The Computer Journal; 1990; 33(2): 126-132.

Dwyer, B. Correspondence about Dromey & Chorvat (1990). The Computer Journal; 1991; 34(1): 72.

Gehani, N; McGettrick, A. *Software Specification Techniques*. Reading, Mass.: Addison-Wesley; 1986.

Ginsburg, S. *Mathematical Theory of Context-free Languages*. New York: McGraw-Hill; 1965.

Gladden, G R. *Stop the life cycle, I want to get off*. ACM Software Engineering Notes; 1982; 7: 35-39.

Godwin, A N; Gore, M B; Salt, D W. *A Comparison of JSD and DFD as Descriptive Tools*. The Computer Journal; 1989; 32(3): 202-211.

Goguen, L A; Meseguer, J. *Rapid Prototyping in the OBJ Executable Specification Language*. ACM SIGSOFT Software Engineering Notes; 1982; 7(5): 75-84.

Goldberg, A; Robson, D. *Smalltalk-80: The Programming Language and its Implementation*. Reading, Mass.: Addison-Wesley.

Goldsmith, M. *occam Transformation at Oxford*. In: Muntean, T. Proceedings of the 7th occam User Group Technical Meeting. Grenoble; 1988; OUG-7.

Gomaa, H A. *A software design method for real-time systems*. Communications of the ACM; 1984; 27(9): 938-949.

Hall, A. Keynote Address. 13th International Conference on Software Engineering; Austin, TX. Los Alamitos, CA.: IEEE Computer Society Press; 1991.

Harrison, W; Magel, K I. *A complexity measure based on nesting level*. ACM Sigplan Notices; 1981; 16: 63-74.

Henderson, P. *Functional Programming, Formal Specifications, and Rapid Prototyping*. IEEE Transactions on Software Engineering; 1986; SE-12(2).

Henderson, P. *Functional Programming: Application and Implementation*. London: Prentice-Hall; 1980.

Hoare, C A R. *Communicating Sequential Processes*. Englewood Cliffs, N.J: Prentice-Hall; 1985.

Hoare, C A R. *Programming: Sorcery or Science?* IEEE Software; 1984; 1(2): 5-16.

Hopcroft, J E; Ullman, J D. *Formal Languages and Their Relation to Automata*. Reading, Mass.: Addison-Wesley.

Hughes, J W. *A Formalization and Explication of the Michael Jackson Method of Program Design*. Software - Practice and Experience; 1979; 9: 191-202.

Hull, M E C; McKeag, R M. *Concurrency in the Design of Data Processing Systems*. The Computer Journal; 1984; 27(4).

Hull, M E C; O'Donogue, P G; Hagan, B J. *Development methods for real-time systems*. Computer Journal; 1991; 34(2): 164-72.

Jackson, M A. *Information Systems: Modelling, Sequencing and Transformations*. In: Proceedings of International Conference on Software Engineering. Washington, D.C.: IEEE Computer Society Press; 1978.

Jackson, M A. *JSD Modelling: Some Underlying Ideas and their Relationship to Object-Orientation and Data Modelling*. In: Jackson, M S. Proceedings of the seminar series on new directions in software development. Wolverhampton: The Polytechnic Wolverhampton; 1988; 1: 1-8.

Jackson, M A. *Principles of Programming Design*. London: Associated Press; 1975.

Jackson, M A. *System Development*. Englewood Cliffs, N.J: Prentice-Hall; 1983.

Jones, C B. *Systematic Software Design Using VDM*. Englewood Cliffs, NJ: Prentice-Hall; 1986.

Kato, J; Morisawa, Y. *Direct Execution of a JSD Specification*. In: Proceedings of 11th COMPSAC. Washington, D.C.: IEEE Computer Society Press; 1987.

King, D. *Creating Effective Software (Computer Program Design Using the Jackson Methodology)*: Yourdon Press (Prentice-Hall); 1988.

King, M J; Pardoe, J P. *Program Design Using JSP*. New York: Macmillan; 1985.

Laurier, J L. *Problemsolving and Artificial Intelligence*. Englewood Cliffs: Prentice-Hall; 1990.

Lehman, M M. *Programs, life cycles and laws of software evolution*. Proc. IEEE Special Issue on Software Engineering; 1980: 1060-1076.

Lehman, M M. *Software engineering, the software process and their support*. Software Engineering Journal; 1991; 6(5): 243-258.

Lewis, C T. *The Realisation of JSD Specifications in Object Oriented Languages*. Doctoral Thesis: The University of Aston in Birmingham; 1991.

Lientz, B; Swanson, E B. *Software Maintenance Management*. Reading, Mass.: Addison-Wesley; 1980.

Liskov, B; Guttag, J. *Abstraction and Specification in Program Development*. Cambridge, Mass: The MIT Press; 1986.

McCracken, D D; Jackson, M A. *A Minority Dissenting Position*. In: Cotterman W W, ed. Systems Analysis and Design - A Foundation for the 1980's. New York: Elseview North-Holland; 1981: 551-553.

McCracken, D D; Jackson, M A. *Life-cycle concept considered harmful*. ACM Software Engineering Notes; April 1982: 29-32.

McCulloch, W S; Pitts, W. *A logical calculus of the ideas immanent in nervous activity*. Bulletin of Mathematical Biophysics; 1943; 5: 115-133.

McDermid, J. *Software Engineer's Reference Book*. London: Butterworth-Heinemann; 1991.

Milner, R. *A Calculus of Communicating Systems*. New York: Springer; 1980.

Minsky, M. *Computation: finite and infinite machines*. London: Prentice-Hall; 1967.

Neilson, K W; Shumate, K. *Designing large real-time systems with Ada*. Communications of the ACM; 1987; 30(8): 695-715.

Nicholls, D. *Introducing SSADM — The NCC GUIDE.* Manchester: NCC Publications; 1987.

Parnas, D L. *On the Criteria To Be Used in Decomposing Systems into Modules.* Communications of the ACM; 1972; 15(12): 1053-1058.

Parnas, D L; Clements, P C. *A Rational Design Process: How and Why to Fake It.* IEEE Transactions on Software Engineering; 1986; SE-12(2): 251-257.

Partsch, H; Steinbrüggen, R. *Program Transformation Systems.* ACM Computing Surveys; 1983; 15(3): 199-236.

Partsch, H. *Specification and Transformation of Programs: A Formal Approach to Software Development;* Springer-Verlag; 1991.

Peterson, D. *Three Cases of Re-representation in Problem-solving.* Cognitive Science Research Group, Univ. Birmingham.

Poo, C C D. *Representing Business Policies in the Jackson System Development Method.* The Computer Journal; 1991; 34(2): 122-131.

Ratcliff, B. *An inversion capability for the PRESTIGE workbench; some basic issues.* In: Proceedings of COMPSAC 14. Los Alamitos, CA: IEE Computer Society Press; 1990: 623-8.

Ratcliff, B. *Software Engineering: Principles and Methods: Blackwell Scientific*; 1987.

Ratcliff, B; Boyle, M. *The PRESTIGE Workbench: CASE Support for the Implementation Phase of JSD*: Submitted for Publication.

Rayward-Smith, V J. *A first course in formal language theory.* Oxford: Blackwell Scientific; 1983.

Rayward-Smith, V J. *Language Theory.* McDermid, J, ed. Software Engineer's Reference Book: Butterworth-Heinmann; 1991.

Renold, A. *Designing a Music Synthesizer with the JSD Method.* Scientia Electrica; 1988a; 34(4): 3-46.

Renold, A. *Jackson System Development for Real Time Systems.* Scientia Electrica; 1988b; 34(2): 3-43.

Rich, E. *Artificial Intelligence.* New York: McGraw-Hill; 1990.

Rockstrom, A; Saracco, R. *SDL—CCITT specification and Description Language.* IEEE Transactions on Communications; 1982; . COM-30(6): 1310-1318.

Roper, M; Smith, P. *A Structural Testing Method for JSP Designed Programs.* Software - Practice and Experience; 1987; 17(2): 135-157.

Roscoe, A W; Hoare, C A R. *The Laws of Occam Programming.* Oxford University Computing Laboratory, Programming Research Group, Technical Monograph; 1986(PRG-53).

Roscoe, A.W. & Dathi, N. *The Pursuit of Deadlock Freedom.* Oxford University Computing Laboratory, Programming Research Group, Technical Monograph; 1986; (PRG-57).

Rose, J. *A New Rigorous Approach for Modelling and Refining Concurrent Behaviour in JSD Specifications*. Structured Programming; 1992; (13): 11-21.

Royce, W W. *Managing the Development of Large Software Systems: Concepts and Techniques*. In: Proc. Wescon; 1970.

Sanden, B. *An Entity-life modeling approach to the design of concurrent software*. Communications of the ACM; 1989; 32(3): 330-343.

Sanden, B. *Systems Programming with JSP*: Chartwell-Bratt; 1985.

Sanders, J W. *An Introduction to CSP*. Oxford University Computing Laboratory, Programming Research Group, Technical Monograph; 1988; (PRG-65).

Schneidewind, N F. *The state of software maintenance*. IEEE Transactions on Software Engineering; 1987; 13: 303-310.

Sommerville, I. *Software Engineering*. Reading, Mass.: Addison-Wesley; 1991.

Sridhar, K T; Hoare, C A R. *JSD expressed in CSP*. Oxford University Computing Laboratory, Programming Research Group, Technical Monograph; 1985; (PRG-51).

Storer, R. *Data-driven software design using inversion*. Information and Software Technology; 1988; 30(2): 99-107.

Storer, R. *Practical Program Development using JSP*: Blackwell Scientific; 1987.

Sutcliffe, A. *Jackson Systems Development*. Englewood Cliffs, N.J.: Prentice-Hall; 1988.

Sutcliffe, A; Wang, I. *Integrating Human Computer Interaction with Jackson System Development*. The Computer Journal; 1991; 34(2): 132-142.

Swartout, W; Balzer, R. *On the Inevitable Intertwining of Specification and Implementation*. Communications of the ACM; 1982; July: 551-553.

Turner, D. *An Overview of Miranda*. ACM SIGPLAN; 1986; 21(12).

Warhurst, R; Flynn, D. *Validating JSD specifications by executing them*. Information and Software Technology; 1990; 32(9): 598-612.

Wilson, A D. *Programs to Process Trees, Representing Program Structures and Data Structures*. Software - Practice and Experience; 1984; 14(9): 807-816.

Yeung, W L; Smith, P; Topping, G. *A formalisation of Jackson System Development*. In: Third International Conference on Software Engineering for Real Time Systems. London: IEE: 31-9.

Yourdon, E N; Constantine, L L. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Englewood Cliffs, N.J: Prentice-Hall; 1979.

Zave, P. *An insider's evaluation of PAISLey*. IEEE Transactions on Software Engineering; 1991; SE-17(3): 212-25.

Zave, P. *An Operational Approach to Requirements Specification for Embedded Systems*. IEEE Transactions on Software Engineering; 1982; 8(3): 250-269.

Zave, P. Personal Communication; 1991b.

Zave, P. *The operational versus the conventional approach to software development.* Communications of the ACM; 1984; 27(2): 104-118.

Zave, P; Jackson, D. *Practical specification techniques for control-oriented systems.* In: Ritter, G X. Information Processing '89. Proceedings of the IFIP 11th World Computer Congress. Amsterdam: North-Holland; 1989.

Zave, P; Schell, W. *Salient Features of an Executable Specification Language and Its Environment.* IEEE Transactions on Software Engineering; 1986; SE-12(2): 312-325.

Appendix I
Smalltalk listings of subset construction algorithms and related code

```
'From Smalltalk-80, Version 2.3 of 13 June 1988 on 3 June 1992 at
5:13:05 pm'



ImplProcess methodsFor: 'subset construction'

actionBody:  aPosition
   self leaves: [:aLeaf | aLeaf position = aPosition
      ifTrue:
         [↑aLeaf exOps printString]].
   ↑''


alphabet
   | alpha |
   alpha ← Set new.
   self leaves: [:aLeaf | alpha add: aLeaf nodeName].
   ↑alpha


dfa
   "transform into a dfa using a subset construction algorithm
based    on that of Aho et al. 1986"

   | dStates unMarked t positionsForInput u tt |
   tt ← TransitionTable new.
   dStates ← MarkedSet new.
   dStates add: self firstPos.
   [(unMarked ← dStates select: [:s | s marked not]) isEmpty]
      whileFalse:
         [t ← unMarked asOrderedCollection first.
         t mark: true.
         self alphabet do:
            [:input |
            positionsForInput ← t select: [:p | (self nameOf: p)
                        = input].
            positionsForInput isEmpty
               ifFalse:
                  [positionsForInput size > 1
                     ifTrue:   [self   error:   'recognition
difficulty']
                     ifFalse:
                        [u ← self followPos:

positionsForInput asOrderedCollection first].
                  u isEmpty | (dStates hasMember: u)
                        ifFalse: [dStates add: u].
                  tt
                     from: t to: u on: input]]].
↑tt
```

dfaOps

```
   "transform into a dfa using a subset construction algorithm
based    on that of Aho et al. 1986"

   | dStates unMarked t positionsForInput u tt et |
   tt ← TransitionTable new.
   et ← TransitionTable new..
   dStates _ MarkedSet new.
   dStates add: self firstPos.
   [(unMarked ← dStates select: [:s | s marked not]) isEmpty]
      whileFalse:
         [t ← unMarked asOrderedCollection first.
         t mark: true. self halt.
         self alphabet do:
            [:input |
            positionsForInput ← t select: [:p |
                  input , '*' match: (self nameOf: p)].
            positionsForInput isEmpty
               ifFalse:
                  [positionsForInput size > 1
                     ifTrue:   [self   error:   'recognition
difficulty']
                     ifFalse:
                        [u ← self followPos:
                        positionsForInput asOrderedCollection
first].
                  u isEmpty | (dStates hasMember: u)
                        ifFalse: [dStates add: u].
                  tt
                     from: t
                     to: u
                     on: input.
                  et
                     from: t
                     perform: ( self actionBody:
                        positionsForInput  asOrderedCollection
first)
                     on: input]]].
↑ DfaOps
      name: 'D' start: 's1' transitions: tt effects: et alphabet:
self     alphabet finals: 's'
```

```
dfaOpsRecogn
   | dStates unMarked t positionsForInput u tt et positTable
effect  admitseq positPos admitPos alphabet |
   alphabet ← self alphabet reject: [ :name | '*btr' match:
name].
   tt ← TransitionTable new.
   et ← TransitionTable new.
   positTable ← Dictionary new.
   dStates ← MarkedSet new.

   dStates add: self firstPos.
   admitseq ← 'restore '. effect ← ''.
   [(unMarked ← dStates select: [:s | s marked not]) isEmpty]
      whileFalse:
         [t ← unMarked asOrderedCollection first.
         t mark: true.
         alphabet do:
            [:input |
            positionsForInput ← t select: [:p |
                  input , '*' match: (self nameOf: p)].
            positionsForInput isEmpty
               ifFalse:
                  [positionsForInput size > 1
                     ifTrue:

"-- recognition difficulty branch -------------------------------
-"

[u ← self distrUnion:
         (positionsForInput collect: [ :pos | self followPos:
pos]).
positPos ← positionsForInput asSortedCollection first.
admitPos ← positionsForInput asSortedCollection last.
positTable at: u put:
                  (self followPos: positPos).
effect ← 'save' , (self actionBody: positPos).
admitseq ← admitseq , (self actionBody: admitPos) printString]

                     ifFalse:

"-- no recognition difficulty branch ----------------------------
-"

[p ← positionsForInput asOrderedCollection first.
u ← self followPos: p.
effect ← self actionBody: p.
"check if a recognition difficulty can be resolved"
(positTable keys includes: t)
      ifTrue:
         [((positTable at: t) includes: p)
            ifFalse: [effect ← admitseq , effect]]].

" -- update dStates, transition table and effects table ----------
-"

                  u isEmpty | (dStates hasMember: u)
                     ifFalse: [dStates add: u].
                  tt from: t to: u on: input.
                  et from: t perform: effect on: input]]].
   ↑DfaOps
      name: self name start: 's1' transitions: tt
      effects: et alphabet: self alphabet finals: 's'
```

```
distrUnion:  listOfSets
   | uSet |
   uSet ← MarkedSet new.
   listOfSets do: [ :set |
      set do: [ :elem | uSet add: elem]].
   ↑uSet

firstPos
   ↑self className followMap at: #first

followPos:  aPosition
   ↑self className followMap at: aPosition

nameOf:  aPosition
   self leaves:
      [:aLeaf  |  aLeaf  position  =  aPosition  ifTrue:  [↑aLeaf
nodeName]]
```

Smalltalk listings of Class DfaOPs including dismemberment code generation and direct life-history recognition

```
'From Smalltalk-80, Version 2.3 of 13 June 1988 on 3 June 1992 at
5:12:42 pm'

Object subclass: #DfaOps
   instanceVariableNames: 'name start transitions effects alphabet
                           finals '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'JSD-Implementation'


DfaOps methodsFor: 'initialization'

name: n start: s transitions: t effects: e alphabet: a final
s: f
   name ← n.
   start ← s.
   transitions ← t.
   effects ← e.
   alphabet ← a.
   finals ← f


DfaOps methodsFor: 'code generation'

dismemberedStates:  states  inputs:  inputs
   | code |
   code ← ''.
   states do:
         [:state |
         code ← code , ' if state = ' , state , ' then
'.
         (transitions at: state)
            associationsDo: [:assoc |
               (inputs includes: assoc key) ifTrue:
                  [ code ← code , '       if input = ' , assoc key
, '
          then ' ,
                ((effects at: state) at: assoc key) printString , '
state := ' , assoc value , '
   endif;
'         ]]].
   ↑code , '
endif;'
```

```
hardcode
   | code |
   code ← 'case state of
'.
   transitions
      keys asSortedCollection do:
         [:state |
         code ← code , state , ' =>
'.
         (transitions at: state)
            associationsDo: [:assoc | code ← code , '      if
input = ' , assoc key , ' then
         ' ,
((effects at: state) at: assoc key) printString , '
         state := ' , assoc value , '
   endif;
'        ]].
   ↑code , 'endcase;'

DfaOps methodsFor: 'recognition'


recognise:  aList
   "Answer whether or not you recognise <aStream>"
   | state currentSym aStream |
   aStream ← ReadStream on: aList from: 1 to: aList size.
   state ← #s1.
   currentSym ← aStream next.
   [currentSym = #eof] whileFalse:
      [state ← transitions delta: state on: currentSym.
      currentSym ← aStream next].
   ↑state = #s
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "

DfaOps class
   instanceVariableNames: ''


DfaOps class methodsFor: 'instance creation'

name: n start: s transitions: t effects: e alphabet: a final
s: f
   ↑(super new)
    name: n start: s transitions: t effects: e alphabet: a finals:
f
```

Smalltalk listings of Class `OperationTable`

```
From Smalltalk-80, Version 2.3 of 13 June 1988 on 10 June 1992 at
11:10:18 am

Dictionary variableSubclass: #OperationTable
   instanceVariableNames: 'readSet writeSet lastLabel firstRead
                          noTransformation '
   classVariableNames: ''
   poolDictionaries: 'PrestigeConstants '
   category: 'PRESTIGE-Support'
```

```
OperationTable methodsFor: 'accessing'

at: index put: statement
   | possibleIOstmt |

   statement isNil ifTrue:
         [super at: index put: statement.
         ↑self].

   (possibleIOstmt ←statement reduceToSingleIO) isNil
      ifTrue:
         [super at: index put: statement.
         ↑self]
      ifFalse:
         [super at: index put: possibleIOstmt.
         "If statement is a read or write,
         add index to the appropriate set"
         (possibleIOstmt isMemberOf: ESTELreadOp)
            ifTrue: [readSet add: index].
         (possibleIOstmt isMemberOf: ESTELwriteOp)
            ifTrue: [writeSet add: index]]

at: index transf: trFlag
   "This operation returns the code fragment at i, transformed
   as appropriate.  Information necessary to decide on any
   transformations to be performed is obtained by inspecting
   the relevant #DatastreamTable and #TransformationContract"
   | transformation |

   trFlag ifFalse: [^super at: index].

   "If the operation is not a read or write,
   no transformation necessary"
   ((readSet includes: index)
      or: [writeSet includes: index])
      ifFalse:
         [↑super at: index].

   "Consult datastream table for transformation, if absent make it
   read inversion by default"
   "transformation ←#TT at: (super at: index) dataStream
      ifAbsent: [transformation ←#RINV]."
   ↑self at: index withTransformation: #RINV

operationList
   "Produce a sorted list of the textual form of each operation
   for use in a ProcessStructure browser"

   | list |
   list ←OrderedCollection new.
   (operations keys) asSortedCollection do: [ :op |
      list add: ((op printString) , ' : ' , ((operations at: op)
         codeString))].
   ↑list

OperationTable methodsFor: 'private'

at: index withTransformation: transformation
   "choose and call relevant method for transforming a rquested
   operation"
   (readSet includes: index)
      ifTrue:
         [transformation = #RINV ifTrue: [↑self
readInvertedRead].
         transformation = #WINV ifTrue:
            [↑self writeInvertedRead: index]].
   (writeSet includes: index)
      ifTrue:
         [transformation = #RINV ifTrue:
            [↑self readInvertedWrite: index].
         transformation = #WINV ifTrue: [↑self
writeInvertedWrite]].
   "else"
   self error: 'Transformation failed: select debug'
```

determineReaderOfMerge: aString

```
   ((PrestigeConstants at: #JSDdBase) at: #TableMerges)
associations do: [:each | each key asSymbol = aString asSymbol
ifTrue:
   [↑each value descriptor reader]]
```

readSet: rs writeSet: ws lastLabel: ll firstRead: fr noTransformation: nt

```
   readSet ← rs.
   writeSet ← ws.
   lastLabel ← ll.
   firstRead ← fr.
   noTransformation ← nt
```

OperationTable methodsFor: 'transforming'

dispatcherIn: pLang

```
   "Source code for jumping to the current text pointer of the
inverted
   process.  This is inserted at the start of the text of an
inverted
   subroutine"

   ↑'Goto L(SV.TP);  L(1):'
```

readInvertedRead

```
   "Answer the code which ensures that the process will
   wait for the data it requires.  If this is the first read,
   answer the empty string and change the firstRead flag
   to false.  firstRead is set to true by calling initFlags"
   "Smalltalk browseAllImplementorsOf: #initFlags"

   firstRead
      ifTrue:
         [firstRead ← false.
         ↑'']
      ifFalse: [↑ESTELWaitForData new label: self nextLabel]
```

readInvertedWrite: op

```
   "Answer the code which ensures that the process will
   call its reader with the data it requires"

   | statement callee dest |
   statement ← self at: op.
   dest ← ((JSDdBase at: #TableDSs)
            at: statement dataStream token asJSDString)
                  descriptor destination.
   (dest at: 1)
      = $M
      ifTrue: ["MERGE, so find its reader and put its name in
callee"
         callee  ← self  determineReaderOfMerge:  (dest
copyReplaceAll: 'MERGE ' with: '')]
      ifFalse: ["non-merge so put destination name in callee
stripped of the
          'PROCESS ' prefix"
         callee ←dest copyReplaceAll: 'PROCESS ' with: ''].
   (callee sameAs: 'PROCESS')
      ifTrue: ["implementation mode is external write"
         ↑statement]
      ifFalse: ["implementation mode is inverted call"
         ↑(ESTELCallWithData new) callee: callee; recordName:
statement record; datastream: statement dataStream token]
```

withTransformation

```
   noTransformation ←false
```

writeInvertedRead: op

```
   "Answer the code which ensures that the process will
   call its writer for the data it requires"

   | statement |
   statement ←self at: op.
   ↑(ESTELCallForData new)
      callee: ((JSDdBase at: 'Data-Streams')
         at: statement dataStream asUppercase asJSDString) writer;
      recordName: statement record
```

writeInvertedWrite

```
   "Answer the code which ensures that the process will
   wait with the data its reader requires"

   ↑ESTELWaitWithData new label: self nextLabel
```

OperationTable methodsFor: 'label generation'

nextLabel

```
   ↑lastLabel ←lastLabel + 1
```

OperationTable methodsFor: 'initialization'

initialize

```
   self do: [ :c | self removeKey: c].
   readSet ←Set new.          "indices of read statements"
   writeSet ←Set new.         "indices of write statements"
   self initFlags             "label for suspend points and first
read"
```

OperationTable methodsFor: 'enumerating'

collect: aBlock

```
   operations collect: aBlock
```

OperationTable methodsFor: 'removing'

removeKey: aKey

```
   "remove the reference to the operation at key.  Remove any
occurence   of key in the read or write sets"

   readSet remove: aKey ifAbsent: [].
   writeSet remove: aKey ifAbsent: [].
   super removeKey: aKey

OperationTable methodsFor: 'growing'
```

grow

```
   "Increase the number of elements of the collection.  This needs
to    be overridden because Dictionary|grow doesn't allow for the
copying
   of extra instance variables which may have been added by
   subclasses."

   | newSelf |
   newSelf ←self species new: self basicSize + self growSize.
   self associationsDo: [:each | newSelf noCheckAdd: each].
   newSelf
      readSet: readSet
      writeSet: writeSet
      lastLabel: lastLabel
      firstRead: firstRead
      noTransformation: noTransformation.
   self become: newSelf

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "

OperationTable class
   instanceVariableNames: ''

OperationTable class methodsFor: 'instance creation'
```

new

```
   ↑super new initialize
```

Appendix II
Examples of dismemberment generated by the software described in the thesis

The BOOK process dismembered by state

This is a straightforward application of dismemberment. Imagine that it has been decided to implement modules to deal with three phases of the life of a book. The first is at the beginning of its life as it is acquired, classified and made available to lenders, the second is the active part of the life-history, during which time it is being lent, renewed and returned, and the third is the part of its life where it is being disposed of. These three modules are quite disjoint from a scheduling perspective — there is no need to load the code associated with `clssifying` a book when handling a `renew` message, for example. Constructing the DFA yields the following Smalltalk-generated transition table:

```
TransitionTable
   (s78->
      Dictionary
         (swap->s
         sell->s )
   s1->
      Dictionary
         (acquire->s2 )
   s36->
      Dictionary
         (outcirc->s78
         lend->s45 )
   s2->
      Dictionary
         (classify->s36 )
   s45->
      Dictionary
         (return->s36
         renew->s45 ))
```

The tool-generated name of each state is made up of an initial `s` suffixed by a list of positions of the original tree which can be matched from the state. Examination of the states built by the construction algorithm shows which ones need to be included in each of the modules. These can then be generated with the following calls:

```
newBook ←
   bookStructure
      invertedDismemberedStates: #(s1 s2)
      inputs: inputs.
useBook ←
   bookStructure
      invertedDismemberedStates: #(s36 s45)
      inputs: inputs.
disposeBook ←
   bookStructure
      invertedDismemberedStates: #(s78 s)
      inputs: inputs.
```

The code generated for each of these modules is shown below. To make the structural features of the example clear, no assigned operations have been inserted into this text.

```
procedure book_fsm_new(input);
case state of
s1 =>
   case input of
      acquire =>
         -- ops
         state := s2
   endcase;
s2 =>
   case input of
      classify =>
         -- ops
         state := s36
   endcase;
endcase;
end book_fsm_new;


procedure book_fsm_use(input);
case state of
s36 =>
   case input of
      outcirc =>
         -- ops
         state := s78;
      lend =>
         -- ops
         state := s45
   endcase;
s45 =>
   case input of
      return =>
         -- ops
         state := s36
   endcase;
   case input of
      renew =>
         -- ops
         state := s45
   endcase;
endcase;
end book_fsm_use;


procedure book_fsm_dispose(input);
case state of
s78 =>
   case input of
      swap =>
         -- ops
         state := s
   endcase;
   case input of
      sell =>
         -- ops
         state := s
   endcase;
endcase;
end book_fsm_dispose;
```

One possible scheduling strategy to exploit this dismemberment would be to run the modules `book_fsm_new` and `book_fsm_dispose` in batch mode, and run the module `book_fsm_use` on-line.

Purchase Order Problem dismembered by state

The Purchase Order Problem is discussed by Cameron (1988). Inspection of the transition table produced by subset construction shows that the approach taken in the thesis automatically derives the state-based partitioning recommended by Cameron. Below is listed a Smalltalk-generated textual repesentation of the transition table:

```
TransitionTable
   (s4561213->
      Dictionary
         (i->s
         n->s4561213
         o->s4561213
         p->s7891011
         h->s )

   s7891011->
      Dictionary
         (y->s4561213
         v->s4561213
         w->s7891011
         z->s4561213
         x->s7891011 )

   s123->
      Dictionary
         (j->s123
         k->s123
         c->s4561213 )
   s->
      Dictionary( ) )
```

The three modules suggested by Cameron prescribe the behaviour of the process in each of the three states `s123` (M1), `s4561213` (M2) and `s7891011` (M3). These states can then be generated with the following calls:

```
m1 ←
   popStructure
      invertedDismemberedStates: #(s123)
      inputs: #all.
m2 ←
   popStructure
      invertedDismemberedStates: #(s4561213)
      inputs: #all
m3 ←
   popStructure
      invertedDismemberedStates: #(s7891011)
      inputs: #all
```

The outline code for each of these modules, as generated by the tool, is as follows:

```
procedure M1_s123(input, state);
   case input of
      j =>
         -- ops
         state := s123
      k =>
         -- ops
         state := s123
      c =>
         -- ops
         state := s4561213
   endcase;
end M1_s123;


procedure M2_s7891011(input, state);
   case input of
      y =>
         -- ops
         state := s4561213
      v =>
         -- ops
         state := s4561213
      w =>
         -- ops
         state := s7891011
      z =>
         -- ops
         state := s4561213
      x =>
         -- ops
         state := s7891011
   endcase;
end M2_s7891011;


procedure M3_s4561213 (input, state);
   case input of
      i =>
         -- ops
         state := s
      n =>
         -- ops
         state := s4561213
      o =>
         -- ops
         state := s4561213
      p =>
         -- ops
         state := s7891011
      h =>
         -- ops
         state := s
   endcase;
end M3_s4561213;
```

The regular expression `(a b| a c)* da` dismembered by input

This example is interesting because it contains both a recognition difficulty and multiple synonymous leaves (the two cases for which Lewis (1991) was unable to cater). For illustrative purposes, the following operation table will be assumed:

```
{1 → [1], 2 → [2], 3 → [3], 4 → [4], 5 → [5], 6 → [6]}
```

If a process structure representing this expression is stored in a variable `regex`, then a complete text in its DFA-based form can be obtained by evaluating the following statement:

```
(regex dfaOpsRecogn) dismemberedStates: #all inputs: #all
```

The code produced is presented below:

```
case state of
s135 =>
   case input of
      d =>
         [5];
         state := s6;
      a =>
         [save,1];
         state := s24;
   endcase;
s24 =>
   case input of
      b =>
         [2];
         state := s135;
      c =>
         [restore,3,4]
         state := s135
   endcase;
s6 =>
   case input of
         a =>
            [6];
            state := s;
   endcase;
endcase;'
```

Two dismembered fragments are shown below. They are respectively, (i) which deals with all `a` inputs, and so caters for the positing of a particular branch and also illustrates the handling of synonymous leaves; (ii) one which deals with `c` inputs and so caters for the admission of an incorrect posit. The call

```
(regex dfaOpsRecogn) dismemberedStates: #(s135 s24  s6) inputs:
#(a)
```

leads to the generation of the following text:

```
case state of
   s135 =>
      case input of
         a =>
            [save,1];
            state := s24;
      endcase;
   s6 =>
      case input of
         a => [6];
```

```
            state := s
         endcase;
      endcase;
```

The call

```
      (regex dfaOpsRecogn) dismemberedStates: #(s135 s24  s6) inputs:
      #(c)
```

produces

```
      case state of
         s24 =>
            case input of
               c =>
                  [restore,3,4];
                  state := s135;
         endcase;
      endcase.
```