The Realisation of JSD Specifications in Object Oriented Languages

Colin Thomas Lewis

Doctor of Philosophy

The University of Aston in Birmingham

July 1991

The Realisation of JSD Specifications in Object Oriented Languages

Colin Thomas Lewis

July 1991

Abstract

Jackson System Development (JSD) is an operational software development method which addresses most of the software lifecycle either directly or by providing a framework into which more specialised techniques can fit. The method has two major phases: first an abstract specification is derived that is in principle executable; second, the specification is implemented using a variety of transformations. The object oriented paradigm is based on data abstraction and encapsulation coupled to an inheritance architecture that is able to support software reuse. Its claims of improved programmer productivity and easier program maintenance make it an important technology to be considered for building complex software systems. The mapping of JSD specifications into procedural languages typified by Cobol, Ada, etc., involves techniques such as inversion and state vector separation to produce executable systems of acceptable performance. However, at present, no strategy exists to map JSD specifications into object oriented languages. The aim of this research is to investigate the relationship between JSD and the object oriented paradigm, and to identify and implement transformations capable of mapping JSD specifications into an object oriented language typified by Smalltalk-80. The direction which the transformational strategy follows is one whereby the concurrency of a specification is removed. Two approaches implementing inversion — an architectural transformation resulting in a simulated coroutine mechanism being generated — are described in detail. The first approach directly realises inversion by manipulating Smalltalk-80 system contexts. This is possible in Smalltalk-80 because contexts are first class objects and are accessible to the user like any other system object. However, problems associated with this approach are expounded. The second approach realises coroutine-like behaviour in a structure called a 'followmap'. A followmap is the result of a transformation on a JSD process in which a collection of followsets is generated. Each followset represents all possible state transitions a process can undergo from the current state of the process. Followsets, together with exploitation of the class/instance mechanism for implementing state vector separation, form the basis for mapping JSD specifications into Smalltalk-80. A tool, which is also built in Smalltalk-80, supports these derived transformations and enables a user to generate Smalltalk-80 prototypes of JSD specifications.

Keywords

Object Oriented Paradigm, Jackson System Development, Smalltalk-80, Transformations, Implementation, Followsets.

Dedication

This piece of work is dedicated to Dominic, the first born of the author, whose future progress in life will be more difficult than others due to his disability.

Acknowledgements

The author wishes to acknowledge first and foremost the continuous help and support from Bryan Ratcliff whose unique ability to improve English text into a standard suitable for submission as a PhD thesis is most gratefully appreciated. Throughout the four years of this research, the author has come into contact with many people who have all given ideas and suggestions in various ways. These include John Ash and Tim Rowledge of Smalltalk Express, Trevor Hopkins and Mario Wolczko of Manchester University, Michael Jackson, John Cameron, John Cook and Mary Carver of Michael Jackson Systems Ltd (now LBMS), Lt. Commander Steve Dowle of the Royal Navy, Andy Bass and Maeve Boyle of Aston University, and all my colleagues at RARDE Fort Halstead such as John Bendall, Chris Burn, Dave Clenshaw, Kim Groombridge and especially Dave Myles. I must also thank Tony Quigley for allowing me to do this piece of research in the first place and to his descendents Arthur Everett, Robin Piercey and John Lankester. Lastly, I must acknowledge the continuous support from my wife Teresa and two boys, Dominic and Thomas who, over the last four years, have seen very little of their Daddy.

All the work for this thesis was carried out on an Apple Macintosh II. The thesis itself was put together using WriteNow 2.2, Canvas 2.0 and PageMaker 4. The author gratefully acknowledges the skill and effort of John Everest for his help in laying out the text.

This work has been sponsored by RARDE Fort Halstead (now DRA Military Division).

Contents

List of Figures

Scientific men have been repeatedly urged to co-operate in finding the solution of the problems that threaten our times. This is an unauthorised and individual contribution to a subject ... It must not be taken as representing any but the author's own original studies in the subject.

Frederick Soddy

1 Introduction and Overview

1.1 Abstraction, Development Methods and Paradigms

Alan Turing in one of his many lectures described the computer as a "Universal Machine" [*cf.* Brown90]. Today, Turing's vision of the wide use of computers is all too apparent. Computers permeate every facet of the entire social fabric: industry, health, defence, education, transport, etc. However, the growing need for automation and computerised systems in our day-to-day working is now becoming an acute problem; society's demand for new and ever more complex systems is continually increasing in its pace [Sommerville89]. Unfortunately, the supply of reliable, cost-effectively produced systems is not materialising quickly enough to quench this demand.

The approach used in the past for building software systems presented a somewhat unfavourable picture to that of hardware development. Unlike the latter, software development was seen as a non-engineering activity; the result was that products were poorly built, unmaintainable and, in many cases, unwanted [Ratcliff87]. This view of software development has gradually shifted over the last decade. An important factor in this shift has been the concerted attempt by the software community to become a more mature industry. By moving out of a 'cottage industry' approach to building systems into one which could be seen as a more scientifically based engineering activity, the discipline of 'Software Engineering' emerged [Naur76]. Originally, this much needed move was due not to careful planning or prudence on behalf of the software community itself, but as a reaction to the increase in hardware performance. Today, with hardware prices continuing to fall along with attendant increases in performance and reliability, the cost and quality of any new system centre on its software. High software development costs and poor quality can ultimately be attributed to system complexity and our inability to deal with it — complexity in terms of both the scale of the problem to be solved and its intellectual difficulty. Handling system complexity is one of the software developer's greatest challenges. "Controlling software development and maintenance has always involved managing the intellectual complexity of programs and systems of programs" [Shaw80].

The main approach to controlling complexity during software development is the use of appropriate abstraction techniques. The term abstraction relates to any mechanism which helps to reduce complexity by separating that which needs to be seen from that which can be hidden [Shankar80, Aretz82]. As examples, programming languages are abstractions over basic machine level semantics, procedures and functions are abstractions over expressions and statements that capture functionality; modules permit abstraction over data

type behaviours. Different languages provide differing abstraction capabilities, but all abstractions are there to help the programmer express solutions to problems at levels of representation conceptually appropriate to the latter.

Abstraction can be used at all stages of software development inasmuch as the development of any system can be decomposed into separate tasks. Each task applies appropriate abstraction techniques, the results of which are used as inputs to another task. The rigorous application of abstraction techniques in each task can be seen collectively as a *development method.* Development methods partition the complexity of solving a problem into manageable chunks. A good development method provides a criteria and techniques to guide the user from one level of abstraction to another. A path often taken in a software development method follows the lines of creating an abstract specification and then successively refining or expanding the specification to generate a more concrete representation of what is wanted, typically a program or suite of programs. This activity can take many forms, and there are many factors which influence the ease of its execution. Any techniques and tools, the use of which can automate or semi-automate this activity and increase the effectiveness of its application, are to be welcomed.

From a general viewpoint, the categories in which are placed different development methods, or programming languages, say, can be referred to as *paradigms.* A paradigm is a way of perceiving certain phenomena of interest under a specific conceptual framework characterised by a collection of axioms that, together, form a particular 'template' or 'model'. For example, computation is based on state transitions in the procedural paradigm, whereas in the functional paradigm, computation is based on mathematical mappings from inputs to outputs. The next section briefly reviews three particular software development paradigms which underpin this thesis, namely the operational, transformational and object oriented paradigms.

1.2 Three Paradigms

Operational

A distinguishing feature of the operational paradigm is its approach to specification [Agresti86, Zave84]. Stated simply, an operational specification is a formalised abstract specification which in principle can be executed, thus generating the behaviour of the specified system. System specifications are created early on in the overall development process and are the basis from which programmers build the required software. However, many development methods construct system specifications that are partly or completely informal. One constraint this imposes is that it is not possible to see the behaviour of the system until parts of the specification are realised in code, which is normally late on in the development process. On the other hand, the execution of an operational specification gives developers an early preview of how the system will behave when implemented. An operational approach can therefore lead potentially to a reduction in the overall cost of development since validation of functional requirements can take place at an earlier stage than usual.

Of the many development methods available to the systems builder, only a few belong to the operational paradigm; examples include Paisley [Zave82, Zave84a], Gist [Balzer82], Me-Too [Henderson86] and Jackson System Development (JSD) [Jackson83]. Of these, JSD is the one that is in commercial use in the UK, and is the one with which this thesis is concerned with.

Transformational

A transformation is a prescription for the mechanical conversion of one formal representation into another that preserves correctness when the representations are viewed under appropriate interpretations [Balzer81]. An obvious example of a transformational system is a compiler. A compiler takes some high-level programming language source code and transforms it into some target machine code representation. These two representations are completely different in syntactic form but interpretively are the same. The compiler embodies transformational rules which exhibit behaviour-preserving semantics. Thus, the externally observed behaviour of executing the machine code is what one would expect from interpreting the source code specification.

The application of formal transformations to artifacts such as specifications and diagrams to generate wanted implementations is in direct contrast to the 'manual' approach usually taken [Agresti86]. A strategy for implementation which is completely informal tends to rely on the ingenuity of programmers to 'hand generate' code. This approach has its problems as there is often little, if any, systematic foundation to guide the programmer. Transformational systems, however, address "the labour intensiveness of software development by using specialised computer software to transform successive versions of the developing system mechanically" [Agresti86]. Transformational tools are usually found at the implementation end of the system development life-cycle e.g. a compiler, but are rarely found in the domain of specification manipulation. Another hallmark of an operational approach like JSD is that it embodies transformational techniques which enable the system builder to generate target implementations from abstract specifications. An opportunity therefore exists to build tools to automate these transformations.

Object Oriented

Object oriented programming is a relatively recent approach to developing software systems that has become very popular with software developers over the last decade. The encapsulation of both behaviour and state within a single kind of entity which amalgamates all processing activity — namely, the object — is perhaps the most significant departure from the procedural paradigm, in which behaviour and state are explicitly separated into 'active' operations and 'passive' data respectively.

An exact definition of 'object oriented programming' is still being debated, but the following characteristics are usually to be found in the object oriented languages currently available: data abstraction and encapsulation in the form of objects, message passing to achieve processing activity, class hierarchies and inheritance to enable code reuse, polymorphism to achieve more general abstractions and, persistence to manage the retention of objects. These concepts will be elaborated on in a later chapter.

Of the many object oriented languages now available, it is Smalltalk-80 which is considered the 'archetypal' [Cook86], 'quintessential' [Wilson87] and 'purest' [America86] of all. The Smalltalk-80 language and environment popularised the object/message metaphor during the eighties, with the result that many new object oriented languages have been developed, such as Eiffel [Meyer88] and CLOS [Keene89]. Additionally, the software community has seen such potential in this new paradigm that procedural languages such as C and Pascal have had object oriented extensions grafted onto them. It is these 'hybrid' languages which are now driving the interest in object oriented technology. "Interest in the object oriented approach to software development has been kindled by the release of extensions to support the necessary additional functionality, and the emergence of new language variants such as C++" [Cooper90].

1.3 Scope of Research

Aim and Objectives

To date, transformations of sufficient generality exist for implementing JSD specifications in the procedural paradigm typified by languages such as Pascal, Fortran, Cobol, etc. However, no transformations currently exist for mapping JSD specifications into object oriented languages. The main objective of this research has been to rectify this situation. In order to identify appropriate transformations, an exercise in comparing and contrasting the two domains involved was undertaken. The identification of general relationships resulting from the comparison exercise was then used to construct transformations that were incorporated into a useful tool. The tool enables a user to prototype a system by entering a JSD specification description and then automatically transforming the specification by generating a series of Smalltalk-80 classes which can be immediately evaluated to show the behaviour of the specification.

It is important to appreciate that the aim of the research has been neither to develop a new method for building object oriented applications nor to modify a current development method (i.e. JSD) to cater for implementation features such as inheritance that are found in object oriented languages. This fact is emphasised because JSD has recently been classed as a possible object oriented design approach for building systems [Masiero88]. Research into development methods which reside within the object oriented paradigm are being undertaken elsewhere [Booch86, Bailin89, McIntyre88, Pun89]. These methods attempt to identify objects and behaviours in their applied problem domains and then classify these objects to form initial specifications. Other techniques are then applied to these specifications to formulate programs which are generally implemented in some object oriented language.

To summarise, the main aim of this research has been to extend the implementation capability of JSD into the object oriented paradigm by realising the following three objectives:

- Identify the general relationships between JSD and the object oriented paradigm;
- Construct a set of general transformations to map JSD specifications into a non-specific object oriented language;
- Automate the derived transformations as a prototyping tool using Smalltalk-80 as the target environment.

Related Work

During the survey of literature, one piece of work carried out in Japan appeared to be strongly related to this research. The Japanese research has culminated in the generation of a tool to execute JSD specifications directly [Kato87]. The approach taken is to describe a JSD specification in a language called JSL (Jackson System development Language). Specifications are entered graphically into the JDE (Jackson system Development Environment) and then annotated using JSL. The JDE parses JSL and generates an intermediate representation which can then be interpreted. The interpreter is made up of five major parts: message parser, process interpreter, scheduler, user interface and utility routines. An interesting point regarding the interpreter is that it is written in an object oriented style using Common Lisp with Flavors; it currently runs on a TI Explorer machine.

The work described here takes a different approach to that of the Japanese. Basically, a specification is entered using a tool held within the Smalltalk-80 environment itself. The tool automatically transforms the specification into a series of Smalltalk-80 classes, which can then be loaded and run. The major differences between this approach and that of [Kato87] are as follows:

- No new language is involved; JSD specifications are entered graphically and annotated directly with Smalltalk-80.
- Entered specifications are transformed into a series of classes which can then be loaded and run outside of the development tool. Hence, code generated from the tool can be exported and run on any other Smalltalk-80 platform.
- Although Smalltalk-80 is the target environment used, the transformations are appropriate to other architectural variants of the object oriented paradigm.
- Unlike the Japanese system, the implemented specification has no concurrency within it, as this is removed by the transformations.

In short, this thesis is fundamentally involved with general transformational systems targeted at object oriented languages, and it is this which ultimately reflects the major difference between the two works.

Areas not covered

JSD is a comprehensive development method that uses a rich variety of techniques and concepts. As such, this research, could not realistically encompass the whole of JSD. To make the research more manageable, consideration of the following features of JSD specifications has been deferred:

- Backtracking - a specialised technique sometimes used when specifying the structure of a process;
- Network loops - situations in which potential deadlock occurs, i.e. when many processes are waiting for each other;
- Controlled data streams - an additional communication primitive giving tighter control over interaction between two processes.

The reader wishing to explore further the above elements of JSD is referred to [Jackson83, Renold88].

1.4 Thesis Structure

The thesis is divided into seven major chapters. Chapter 2 gives a detailed introduction to object oriented environments. A general historical introduction to the development of the paradigm is presented in the first section. Next follows a brief review of all the concepts to be discussed throughout the chapter followed by a description of what objects are. There then follows three sections discussing other aspects of object systems — object characteristics, object organisation (inheritance) and finally object communication (message passing).

Chapter 3 presents the JSD method but from a perspective not usually adopted. Although the presentation of the method follows the usual route of modelling, specification and implementation phases, the activities undertaken in each phase are compared with the object oriented paradigm. This presentation gives a firm basis on which to develop the wanted transformations, presented in the following three chapters.

Chapter 4 discusses transformational systems and the transformational approach to implementation. A novel transformational approach specifically for the Smalltalk-80 system is presented, which involves manipulating the run-time stack of the virtual machine. In order to understand this, a brief overview of the Smalltalk-80 system is included. Finally, it is shown that although this approach works well, it is not a general transformation and so has to be abandoned.

Chapter 5 presents a more general transformational approach to that given in Chapter 4. The approach is based on the use of an abstraction called followsets. Basically, process structures are transformed to an alternative representation to which the standard techniques of inversion and state vector separation can be applied. It is shown that the usual single pass approach to implementation (inversion and state vector separation are usually applied at the same time) has become a two phased approach by having to first destructure process structures.

The penultimate chapter presents a description of the specific implementation techniques for realising the transformational approach described in Chapter 5 in Smalltalk-80. A prototyping tool, supporting those transformations including some of the techniques applied in the user interface, together with features provided to make the developer's task more easy, are also described.

The concluding chapter presents a brief resume of the main points in the thesis, examines the extent and success of the work accomplished, and discusses the support tool's uses and limitations. The chapter ends with some remarks on how the current work could be usefully elaborated as part of a future research package.

The beginning of wisdom for a programmer is to recognise the difference between getting his program to work and getting it right. A program which does not work is undoubtedly wrong; but a program which does work is not necessarily right.

Michael Jackson

2 General Characteristics of Object Oriented Languages

2.1 Historical Context

General

The cornerstone of object oriented programming is the encapsulation of abstract behaviour [Tsichritzis88], and the reuse of software components [Meyer87]. Because of these two characteristics, object oriented programming can be classed as a 'packaging' paradigm [Cox86]. Its claims of improved programmer productivity [Wilson87] and easier program maintenance make it an important emerging technology for building complex software systems [Pascoe86]. The historical development of the object oriented paradigm is quite difficult to trace, as there is no one person or organisation solely responsible for its development. Most of the software community attributes the birth of object oriented programming to the Simula language [Dahl66] and its commercial popularisation via Smalltalk-80 [Stefik86, Harland84]. However, the historical transition from Simula to Smalltalk-80 is complicated by the fact that many sources have, in some small way, contributed to the paradigm's development.

The first reference to ideas about objects can be found in work undertaken *c.* 1962/1963 by Ivan Sutherland at MIT [Sutherland63]. Sutherland produced a system called Sketchpad, which enabled a user to enter graphically engineering drawings using a light pen. He " ... believed that the pictures on the screen should be a representation of some meaningful structure inside (the computer), and so Sketchpad was the first object-oriented programming system: it had objects, instances and classes" [Kay87]. Sutherland's insistence on having internal data structures for each image on the display allowed users to make multiple copies of objects without having to redraw them individually.

The next development took place in Europe with an extension to the Algol 60 language undertaken at the Norwegian Computing Centre by Ole-Johan Dahl and Kristen Nygaard, who produced a language specifically for simulation programming and system description. That language was Simula-1, designed between 1962-64 and available in 1965 [Nygaard86]. Simula-1's departure from Algol-60 was the introduction of *activities* (classes) enabling programmers to build data abstractions. Instances of these activities, called *processes* (objects), could be generated in a similar fashion to copying shapes in the Sketchpad system. The second major release of the language, Simula-67 [Dahl66], renamed activities and processes to *classes* and *objects* respectively. The significant addition to the language was "the inheritance concept, that ... solved the memory resource problem by making commonality between

different classes explicit" [Goossenaerts88]. Most of the object oriented programming community see Simula-67 as the start of the object oriented revolution [Thomas89]. However these new ideas did not gain widespread attention until the development of Smalltalk [Tesler84] and as such have made Smalltalk "... the principal object oriented language" [Diederich87]. As Smalltalk-80 plays a major role in this research, a separate historical description of it can be found in the next section.

Since the commercial availability of Smalltalk-80, other object oriented programming languages have emerged. Most of these languages are hybrids [Baines89], retaining their original constructs as well as having object oriented extensions. Possibly one of the more interesting of these hybrid languages (the first being Flavors, the Symbolic's extension to its Zeta Lisp [Moon86]) was Clascal, an extension to Pascal, developed at Apple Computer Corporation USA which first appeared on their Lisa series of microcomputers. Clascal went through several modifications (undertaken by N. Wirth) and was renamed Object Pascal, the language since used for most software development on the Apple Macintosh series of machines. The success of Object Pascal lay in the fact that Apple, as well as providing developers with an object oriented language, also provided a class hierarchy called MacApp [Schmucker86] for reuse, similar to that provided by the Smalltalk-80 environment.

During the last decade, the popularity of the object oriented paradigm has rapidly increased, and is becoming the "... structured programming of the eighties" [Rentsch82]. With four European conferences and five world conferences (the fifth was a joint European and world conference held in Autumn 1990), research in this area is very active. The emerging favourite programming language currently seems to be C++ [Baines89], a hybrid of the C language. C++ was developed at Bell Labs (where C itself was born) by Bjarne Stroustrup. The project in which C++ emerged originally set out to correct some of the shortcomings the C language was perceived to have. The language was released in 1984, shortly followed by the *de facto* standard C++ text [Stroustrup86]. Today, C++ compilers can be found on most Unix workstations and many PCs.

Smalltalk-80

The development of the Smalltalk language has its origins in the Flex (FLexible EXtendable) language developed by A. Kay [Kay69]. Kay had been exposed to both Simula and Sketchpad [Sutherland63], and incorporated their object oriented mechanisms into his language. His piece of original thought was to devise a system to enable these objects to undertake some kind of processing activity and communicate with each other. Thus, the idea of sending messages to objects was conceived [Nelson80].

Smalltalk's development was started during 1971 by the Learning Research Group headed by Alan Kay at Xerox Parc and went through four major stages. The impetus for Smalltalk was to "... support children of all ages in the world of information" [Ingalls78]. This goal was driven by "... the desire to make simple things VERY simple and complex things VERY possible" [Deutsch89]. The first Smalltalk, Smalltalk-72 [Schoch79], was developed between 1971-75 and included many features found in today's Smalltalk platforms, one of the more notable of these features being 'Turtle' graphics that

came from the Logo language. Logo is a dialect of LISP and was developed under the direction of Seymour Papert at MIT during the late 60's. Logo's ability to produce (spirograph like) graphics easily " ... largely accounts for the adoption of turtle graphics as a subsystem of languages such as Smalltalk, Pascal, and even some implementations of PILOT" [Lawler82]. At the same time as Smalltalk-72 was being developed, the use of objects emerged in the parallel processing community. The idea was to represent concurrent activities as actors where message passing was used for concurrent computation. The actor abstraction was developed by Hewitt at MIT, and has since been formalised by Agha [Agha86]. It has been the basis for languages such as Plasma [Hewitt77], Act1 [Lieberman87] and ABCL/1 [Yonezawa86]. It is hard to determine exactly who influenced whom at this period, but it is probably true to say that there was much 'cross-fertilisation', as evidenced by the acknowledgements in Hewitt [*ibid*].

The next major incarnation was Smalltalk-76 [Ingalls78]. This differed from Smalltalk-72 in that it had a fixed syntax and so could be compiled into a byte code representation for efficient interpretation. (In Smalltalk-72, "each class was responsible not only for its own behaviour and state definition, but even for parsing the token stream that followed a mention of an instance" [Deutsch89]). Probably the most significant introduction to Smalltalk-76 was " ... the creation of the Browser by Larry Tesler ... The Browser was a startling innovation at the time, and is still superior to the macro-scale navigation facilities in most environments" [Deutsch89]. It was during this period that the concept of objects was being used by the AI community for representing knowledge, via Minsky's frames [Fikes85]. Smalltalk-78 had no changes of documentable significance to it, but did mark the first attempt at implementing the virtual machine on hardware (Intel 8086) other than Xerox's own.

The Smalltalk used today is essentially the final manifestation of the Smalltalk project, namely Smalltalk-80 developed in 1979-80. Two important developments were reflected in this latest version. The first was the introduction of 'metaclasses' and the second was the 'Model-View-Controller' (MVC) architecture for building interactive applications. The first public airing of Smalltalk-80 was in August 1981 with the special issue of the Byte magazine [Byte81]. Then, in 1983, the 'Blue Book' was released, which has since been the *de facto* definition of the Smalltalk-80 language and its implementation [Goldberg83]. As an aside, one interesting offshoot from the release of the Smalltalk literature was an attempt by Professor David Patterson of Berkeley to implement the Virtual Machine on an optimised RISC-based processor. The processor was called SOAR (Smalltalk On A RISC). Some of the results of this research can be found implemented in the current SPARC RISC processors in Sun workstations [Sun89].

The next section will discuss object oriented systems by highlighting those features which 'most obviously' provide contrast with the procedural approach. This is necessary because the task of describing object oriented systems is not clear cut; the consensus of opinion regarding their essential 'ingredients' has not, as yet, been solidified, possibly due to the paradigm's continued evolution [Cox86, Rentsch82, Pascoe86][1]. Few of the references cited in this section agree on what characteristics constitute the 'essentials' of object oriented systems. However, much research into the semantics of these systems has been undertaken [Minkowitz87, Wolczko88, Yelland89], providing a basis for a better formal

[1] The ECOOP'89 meeting still supported this position.

exposition of the subject in the future. Unfortunately, many new software packages released for the PC market claim to be object oriented when it is patently obvious that they are not. Bjarne Stroustrup, in his invited paper at ECOOP'87, candidly expressed this situation in the following syllogistic fallacy [Stroustrup88]:

Ada is good.
Object oriented is good.
Ada is object oriented.

Some see a danger in this relatively new technology becoming a 'bandwagon' like the structured programming revolution did two decades ago [Beck89, Rentsch82].

2.2 *Object Oriented Concepts*

Introduction

Most object oriented environments are composed of a hierarchy of *classes*. Classes within hierarchies *inherit* attributes and functionality from other classes higher up in these hierarchies[2]. A class is the collective description of a group of similar *objects*. An object is represented by internal local variables or *instance variables* and a set of operations manipulating the instance variables, usually modularised into procedures or *methods*. Unlike standard procedural programming, there is no implicit separation between data and operations. "Programs are not primarily partitioned into procedures and separate data. Rather, a program is organised around entities called objects that have aspects of both procedures and data" [Bobrow83]. All objects in a system are unique and as such have an *identity* which can distinguish one object from another, even if they are instances of the same class both having identical states. Object identity enables objects to refer to themselves in computations, usually via a variable called *self*. Classes and objects usually *persist* within an environment independently of the changes to the state of the object until they are no longer needed [Danforth88].

Since an object functions as an *abstraction* and *encapsulation* mechanism, preventing external access to its state, processing activity is accomplished by sending a *message* to an object and asking it to carry out one of its operations. An object oriented program can thus be viewed dynamically as a collection of objects sending messages to each other. The set of messages to which an object can respond is known as its *protocol*. Many objects may have a similar protocol specification. However, each object can exhibit different behaviour for the same message; this being a manifestation of *polymorphism*.

In a system development context, classes of objects are usually used to represent real world entities, providing a very powerful modelling capability. However, if all objects in a system are *first-class* [Strachey67], they can also be used to represent the structure and processing activity of the object oriented system itself. Such a potential *reflective* architecture enables powerful exploratory programming environments to be devised, since users are able to manipulate the fundamental processing and structuring mechanisms of the environment itself [Maes87].

---

[2] Alternatives to classes and inheritance are briefly mentioned at the end of this chapter.

A 'good' abstraction is one where relevant information is presented to a user and emphasised, but information which is immaterial is completely opaque [Shaw80]. Procedural high level languages provide behavioural abstractions in the form of procedures, functions and primitive operations, and control abstractions in the form of loops and conditionals. However, procedures and functions "... while well suited to the description of abstract events (operations), are not particularly suited to the description of abstract objects" [Guttag77].

Traditionally, variables hold onto data values to be manipulated by a program. The 'type' of a variable simply reflects the values it can hold, classifying sets of values by their representational structure. Although types in languages such as Pascal and Algol promote a more secure programming approach [Harland88], they do not properly furnish any means of representing external entities. Data abstraction, as its name implies, is an abstraction over how a data object is to be manipulated during program execution. Instead of viewing similar data objects merely as a set of values all having the same type, data abstraction perceives data objects by their behavioural characteristics, i.e. what can be done with them. Once data is viewed in this way, abstractions reflecting external realities can be built. Moreover, the specification of a data object's behaviour can be presented without the need to know how that behaviour is implemented. When data objects or data abstractions have a representation-independent specification of what they can do, they are classed as *abstract data types* [Guttag77]. The first language to support the association of abstract operations with the entities on which they were defined as a 'class' construct was Simula-67 [Rowe81].

In order to use a data abstraction, it is not necessary to understand how the abstraction has been implemented; the user is interested merely in its behaviour as described in its specification. Implementation of this behaviour is usually hidden, giving rise to data abstractions being used as a useful form of modular programming [Snyder86a]. Since changes to the implementation of a data abstraction's behaviour will not effect users of it, changes to one part of a system should not effect another part. However, if the internal representation of a data abstraction can be manipulated without using the specified operations, because a language does not enforce this, then the modularisation breaks down.

"Encapsulation is a technique for minimising interdependencies among separately written modules by defining strict external interfaces" [Snyder86a]. Encapsulation is an enforcement or 'policing' mechanism, typically used in conjunction with data abstractions. This mechanism guarantees that access to an abstraction is solely via its specified operations. Encapsulation in software production is used to reduce what is known as the 'ripple' effect when changing some part of a system. Once the only access to an abstraction's state is via its specified operations, there is less chance that changing one part of a system will adversely affect another. Change always becomes local to the encapsulated entities, reducing the dependencies abstractions have on each other. Addition of new data abstractions does not have 'knock-on' effects throughout the system. As system size increases, the importance of encapsulation becomes paramount [Stroustrup88].

In an object oriented environment, objects can be viewed as encapsulated data abstractions. An object's state (set of instance variables) is in scope only to the operations (methods) belonging to that object [Wolczko88] and can affect what an operation returns on completion of its execution. An object's state imparts to the object the property of being mutable, i.e. it can change over time. This contrasts with the other kind of entity used in programming languages, that of a value which is immutable and atemporal [MacLennan82]. Objects are created during the course of a program's execution and, while in existence, their state can change. In conventional programming languages such as Pascal, mechanisms which reflect this type of dynamic behaviour include pointer types; most other entities in the language are values which are neither created nor destroyed (Integers, Reals, etc.).

Object mutability incurs problems in memory management. Objects no longer of use need to be removed from system memory. Two possible options are either to let the programmer remove them or let the language and environment do it automatically. The former is less attractive as the programmer has to understand the whole system in order to manually release an object; programmers have to guarantee that no other part of the system is using the object about to be released. However, this manual approach is adopted by many of the hybrid object oriented programming languages such as Object Pascal and C++. The automatic approach involves a technique called *garbage collection* [Cohen81]. It is beyond the scope of this thesis to discuss different garbage collection algorithms, but three such algorithms, Mark and Sweep [Cohen81], Reference Counting [Deutsch76] and Generation Scavenging [Ungar84] are probably the most well known. All garbage collection techniques attempt to identify obsolete objects and then reclaim the space they occupy. Garbage collection inflicts some performance degradation on the system, but the increased programmer productivity gained is probably worth the cost [Cox86, Pascoe86].

Objects which share the same behaviour are grouped together into a class. In most object oriented programming languages, classes differ from objects in that a class is used as the repository for the description of similar objects: "... the class concept ... is used to express the behaviour of a set of objects which share the same semantics operating on the same attributes" [Cointe87]. Classes have two roles, that of a general type declaration mechanism for similar objects [Florentin85, Thomas89], and as an organisation mechanism (hence classes are more than just the types of a language). "Each class defines a package of behaviour; this can either be instantiated to create an object that conforms to that behaviour,

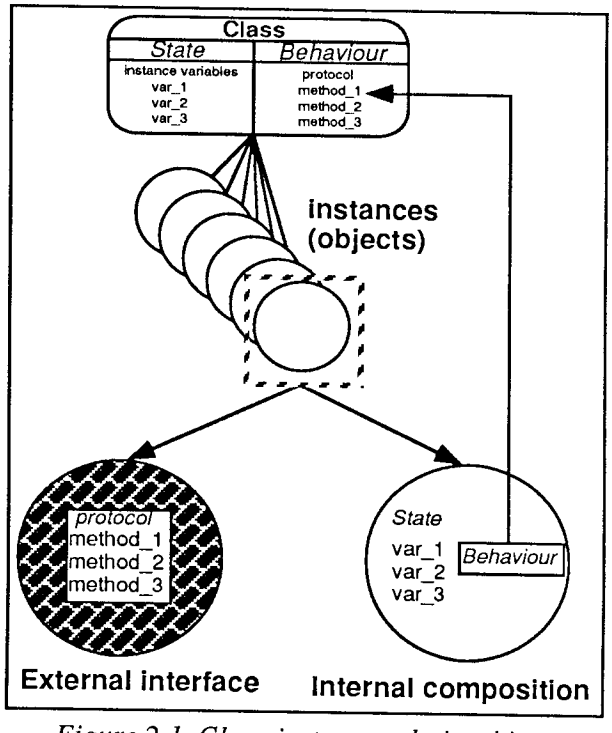


Figure 2.1. Class-instance relationship.

or inherited to define new, but related behaviour." [Wolczko88a]. An object is an *instance* of a class. Objects instantiated from the same class share the same operations and as such have common behaviour (see Figure 2.1 previous page); individual states associated with objects from the same class reflect the differences between them. However, other languages, in particular Smalltalk-80, view classes as instances of an even higher abstraction called a *metaclass*. In environments such as these, the distinction between classes and instances of classes becomes blurred.

Although abstraction and encapsulation are probably the most important attributes of the class-object/instance paradigm, they do not enable a complete differentiation to be made with other programming language constructs which also employ abstraction and encapsulation, such as the Ada package [Booch83], Clu clusters [Liskov80] and Modula-2's modules [Wirth85]. However, these constructs are not treated as 'first-class' citizens, nor do they have a unique 'identity' that 'persists'. To extend this informal characterisation of objects, the attributes of *identity*, *persistence* and *citizenship* will now be discussed.

2.3 *Identity, Persistence and Citizenship*

Object Identity

Objects populating an environment must be distinguishable in order to be usable. Distinguishability amongst objects can be viewed in terms of: similarity, equality and identity. Similarity is where two objects have been instantiated from the same class and hence share the same behaviour and structure. Equality is when two objects have been instantiated from the same class (similarity) and have the same internal states (i.e. the instance variables contain the same values). Identity is where two objects are actually the same object. Instances of objects of the same class can be differentiated only if each object has a distinguishing feature which is not based on the values or state which it possesses [Tsichritzis88]. *Object identity* is an inseparable part of an object once it is brought into being and should remain part of the object no matter how much the environment in which it resides changes [Shilling89]. Objects should have a unique identity during their entire lifetime [Kersten86].

Khoshafian and Copeland [Khoshafian86] highlight two domains involved in supporting identity; representational and temporal. A distinguishing feature of different languages is the degree to which they support representational identity. The strongest form is where it is an inseparable part of an object and built into the language itself. A less strong form is where objects are identified by user-defined name. Finally, with the weakest form, objects are identified simply by the values they represent. Temporal identity can be used to distinguish languages supporting representational identity (in whatever degree) by indicating for how long that representation remains: purely for the duration of program execution, over many executions, or even after the environment in which the object resides is completely restructured. A consequence of languages supporting strong temporal identity is that objects must survive longer than the programs or activities in which they participate. This leads to the another attribute of objects: *persistence.*

Persistence is an abstraction over the time that a piece of data is required and usable [Morrison87] and underlies many current object oriented programming languages [Wolczko88]. In conventional programming systems, the persistence of each data item has to be specifically handled by the programmer via the two persistent mediums available, files and databases. In Pascal, for example, the programmer has only one structured data type for persistent objects, the file type. Other types in the language describe purely transient objects, which last for at most the duration of a program execution.

Extensions to classic von-Neumann architecture include virtual memory and secondary storage. When a program is placed upon this view of storage, the physical properties of the store are not fully abstracted over, so the user is still offered two types of storage medium: program variables and files. Data within a program is therefore either short term or long term, whether data persists or not should be completely invisible to the user. This abstraction over data "... has certain disadvantages. Firstly in any program there is usually a considerable amount of code, typically 30% of the total, concerned with transferring data to and from files or a DBMS" [Atkinson83]. Also, to facilitate the long term storage of structured data, a transformation often has to be employed to 'flatten out' that data. Furthermore, files do not (usually) have any scoping within programming languages and are treated as global entities; this leads to the data stored being insecure. The programmer (when developing a program) has to decide during the design process what attribute the data is to have during its use — to be transient or persistent. This is an additional burden in program development. "Having this two-level storage structure imposes a heavy burden of storage management on the programmer and causes large space and time penalties due to the translation and transfer of information between computational and backing storage media" [Harland88]. Finally, most programming languages only have abstractions to manipulate internal store. Secondary storage manipulation is usually left to either the environment's operating system or some DBMS. From a language design point of view: "If a high-level language can organise RAM it should be able to do the same for disk" [Cockshott84].

"In a persistent system the use of all data is independent of its persistence" [Morrison87a]. There is a consistent view placed on all objects, the persistence of which does not effect their use [Thatte86]. Within a persistent store, objects can outlive the programs which create and use them. Objects are kept for as long as they are usable. Persistent stores place an additional abstraction over the physical storage mediums which a computer system uses; hence, there is no distinction made between file store or internal memory. The advantages of using a persistent store are many: any type of data, ranging from complex tree structures to sequential processes can be made permanent. With the emergence of persistent storage systems, " ... the very concept of a file is rendered almost redundant" [Harland88]. An example environment that offers a form of persistent behaviour is the Symbolics Lisp machine. Here the persistent mechanism used is called a 'world'; when a user restores a world, everything is in exactly the same state as when the world was last saved. Worlds are implemented, to a large extent, on conventional file store architectures. Because of this, the persistent nature of worlds breaks down when a user wishes to communicate outside of a world environment; the communication mechanism inevitably has to be

something other than an object [Merrow87]. This same restriction can be found in Smalltalk-80, which has a weak form of persistence in the form of its 'virtual image' in which all objects persist. This enables object states to be preserved between sessions of execution. However, in order to guarantee such persistence, the virtual image has to be explicitly saved ('snapshot') by writing to disc a complete binary copy of the dynamic memory [Low88, Straw89]. In both the case of a Symbolic's world and Smalltalk-80's virtual image, the persistent mechanisms handle single users only and so objects cannot be shared. To combat this deficiency, multi-user object oriented database management systems (OODBMS) have emerged: Gemstone [Maier86, Penney87] and VBase [Andrews87, Duhl88] provide environments in which to store and share objects amongst many users; they attempt to blur the distinction between database and general programming, providing a unified powerful programming environment.

Citizenship

'Citizenship' and 'civil-rights' would appear strange terms when discussing programming languages. However, when designing a new programming language, there are several theoretical design principles which can be used to produce a 'good' language [Harland84, Harland84a]. One of these principles is known as 'Data Type Completeness' [Reynolds70]. This design principle states simply that all values that the language supports, no matter what that type, should be able to be passed to functions, returned from functions, assigned to variables and form components of data structures. There should be no discrimination as to what can be done to the values in the language regarding their movement; all citizens should have equal civil-rights. Discrimination gives rise to different entities in the language being either *first-class* or *second-class* citizens [Strachey67]. Progress in the expressive power of programming languages can be attributed to making abstractions first-class [Friedman84]

When classes are elevated to first-class status in an object oriented language (as they are in Smalltalk-80 and CLOS), it is primarily this attribute which distinguishes classes from types in standard languages [Sakkinen89]. The following example, taken from the Pascal language illustrates the difference between first and second class citizens:

```
program IncorrectProgram;
      label 10, 20, 30, 40;
      begin
            goto <some-expression>;
      20:
      end.
```

Labels in the language are clearly second-class as they can neither be computed, passed around or manipulated in any way. Also, it is possible in Pascal to pass as a parameter a procedure or function to a procedure. However, since procedures and functions cannot be assigned to variables or returned as values from procedures or functions, they are second-class [Atkinson87]. To exploit the full potential these objects offer in a language, no restrictions should be placed on their potential usage; they should always have first-class status [Atkinson85].

A corollary of objects being first-class is that a language becomes extensible [Foote89]. If new objects are defined in the language, and they have the same status as language-supplied types and objects,

then the result is a superset of the original language. With first-class objects, programmers can create new structures in the language that do not necessarily have to be objects representing entities external to the language system. When a language system allows objects to represent facets of the computational system in which they reside, that system is said to be *reflective*. "Computational reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation" [Maes87]. Reflective architectures help in the organisation of a system's internals. Possibly one of the best examples of reflection at work is in the development of program debuggers. Conventional, non-reflective programming architectures cannot provide debugging facilities from within themselves; debugging a program usually involves invoking a system debugger which is part of the operating system. Also, programs have to be recompiled to include 'debug' information in order for the debugger to operate. Debugging a program in a reflective architecture is more straightforward — when a reflective system is running, the runtime state of that system is represented by user-accessible first-class objects. This enables debugging and tracing facilities to be built from within the language itself, making recompilation and the inclusion of debug information unnecessary. Debuggers in semi-reflective systems, such as CLOS and Smalltalk-80, are superior to those offered by languages such as Pascal and Fortran. However, there are very few completely reflective programming environments available other than 3-KRS [Maes87a] and possibly OBJVLisp [Cointe87]. Nevertheless, programming language designers are beginning to realise the importance of reflection [Wolczko88a].

2.4 *Organisation: Inheritance*

Most object oriented languages use the class as the templating mechanism for instantiating new objects. The other main function of classes is an organisational facility via the *inheritance* mechanism, which supports software reuse [Strom86, America86]. "Inheritance enables programmers to create new classes of objects by specifying the differences between a new class and an existing class instead of starting from scratch each time. A large amount of code can be reused in this way" [Pascoe86]. When a new class is defined, it is always a *subclass* (and specialisation) of another class, called its *superclass*. Inheritance makes it possible to include behaviour of superclasses in new subclasses being defined as well as new behaviour. For the subclassing mechanism to function, classes are usually arranged in a hierarchy in which every object is an instance of just one class; this is *single* inheritance. *Multiple* inheritance is a generalisation of single inheritance [Madsen88] that allows a given class to have more than one immediate superclass (the inheritance hierarchy is not a pure tree); see Figure 2.2 overleaf.

Inheritance promotes the sharing of information between classes and eases the modification of a system, since the behaviour to be changed is always to be found in one place; changing it will affect all classes which inherit this behaviour. "Sharing makes for a usable system by facilitating factoring, the property of one thing being in only one place. Successful factoring produces brevity, clarity, modularity, concinnity, and synchronicity, which in turn provide manageability in complex systems" [Rentsch82]. Another important aspect of sharing concerns the class-instance relationship. "A modification of a method in the class is automatically reflected in the behaviour of every instance of the class" [Kafura89].

This change applies even if the method is in a class higher up in the class hierarchy. Sharing of this kind offers great leverage in modifying the behaviour of objects, as this can be done (in some object oriented environments) while the system is running.

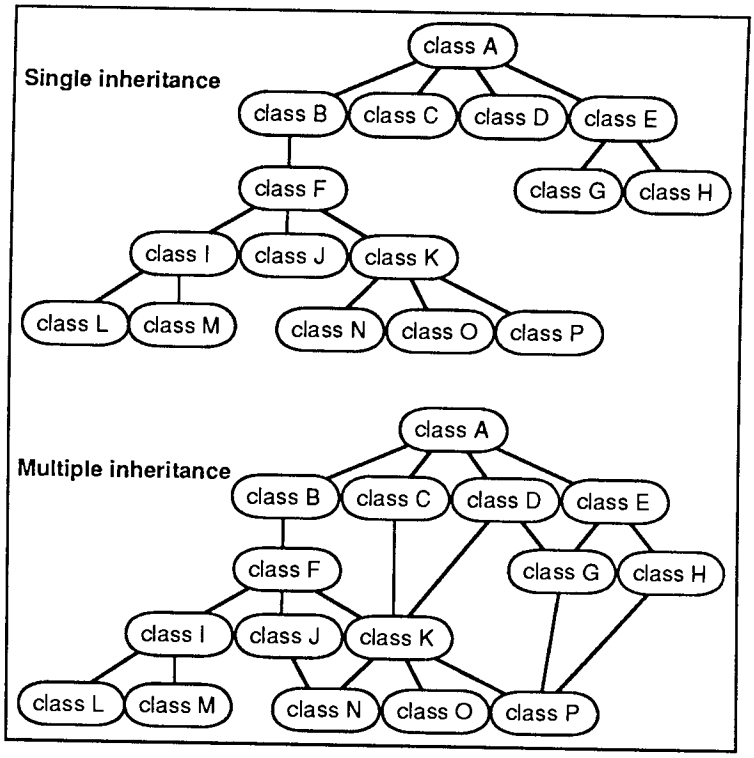


Figure 2.2. Single and multiple inheritance.

Codefactoring ensures that a piece of code has to appear, and hence be written only once in the system; this facilitates software code management as the programmer has to write less code [Ingalls81]. For example, if it were required that all objects be able to give a printed representation of themselves, and inheritance was not available, then every object would have to implement a specific print method — there would be as many print methods as there were types of object in the system. Inheritance and code factoring overcomes this by enabling the programmer to define a print method to be implemented at the top of the class hierarchy. In a single inheritance architecture, every object in the system inherits the default behaviour of the class at the root of the tree and so all objects, whatever their class, can give some printed representation of themselves.

Inheritance is an excellent tool for developing reusable software components [America87] and an excellent design mechanism for building taxonomies of classes [Wegner86]; it allows both the specification of a class's behaviour (organisational aspects) and the structure or implementation of that behaviour (software reuse) to be inherited at the same time. However, these two aspects can lead to confusion [Sakkinen89] as they are "... more or less mutually exclusive ..." [Hendler86]. A new class may have to be defined which has no logical connection with its parent, except that it contains some behaviour which can be reused [Madsen88]. Pun and Winder [Pun89] describe these two aspects of inheritance as 'strict' and 'non-strict' inheritance , where non-strict implies the inheritance mechanism is used specifically for code reuse, whereas strict inheritance is used for design purposes. Some object oriented systems allow programmers to define classes which are not allowed to be instantiated; they are there merely for subclasses to inherit their behaviour [Sandberg86, Borning87]. These classes, known as *abstract* classes, are where much of the system code factoring (methods being moved up the inheritance hierarchy) reside.

Classes share their behaviour with their subclasses. However, if this effect is not wanted then behaviour in specific classes can be *overriden*. Overriding a method in a class means defining a new method but using the same name as one defined in the class's superclass chain. Encapsulation offers

object architectures the ability to define many methods having the same name in different classes. Each method name in the system does not have to be globally unique, but only within the class in which it is defined. However, although encapsulation and inheritance together form a potentially polymorphic architecture (see later), the two mechanisms are not completely orthogonal. A class has two interfaces, a specification of what its instances can do — its instance interface, and a specification of the behaviour that can be inherited by its subclasses — its subclass interface. Subclassing in many object oriented programming languages gives free access to the internals of classes higher up in the hierarchy, thus violating the encapsulation mechanism (CommonObjects [Snyder86] is an example of a language where this does not happen). Many object oriented languages do not encapsulate their class's instance variables from the subclassing mechanism, leading to a weakening of "... one of the major benefits of object oriented programming, the freedom of the designer to change the representation of a class without impacting its clients" [Snyder86a]. Encapsulation guarantees that users of an object (instance interface) will not be affected by a change in its representation. However, inheritance undoes this by making a class's definition globally visible to all its subclasses (see Figure 2.3).

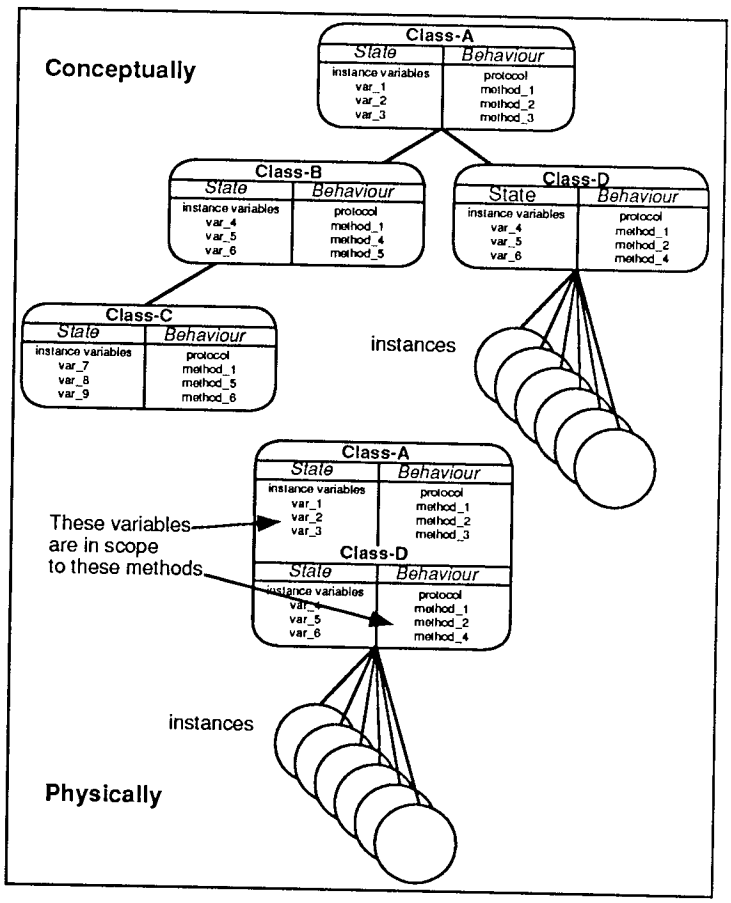


Figure 2.3. Visibility in class hierarchy.

Changing the representation of instance variables in a superclass could have drastic effects on all subclasses which use those instance variables. Subclassing and inheritance force a class's specification to be 'open' all the way down the hierarchy. This openness leads to much confusion [Taenzer89], as an understanding of how a class has derived its behaviour is needed when a new subclass is to be defined: has a method in the parent class been newly defined, inherited or overriden? An example of this confusion occurs in the Smalltalk-80 environment. A class to display information on the screen should usually be made a subclass of the class `View`. However, the novice programmer will not realise that to get this new view class working, the method `displayView` defined in class `View` has to be overriden. The programmer has to understand how `View` is implemented in order to make a successful subclass of it.

The problem in overcoming instance variable subclass visibility is to make sure that all access by a subclass to instance variables in its superclass chain is via message protocols provided by its parents [Rochat86]. This technique has been adopted in building the software supporting this thesis. However,

although this partially resolves the problem, it unfortunately introduces another. Providing access to a class's instance variables via its message protocol makes those instance variables visible to all users of instances of that class, i.e. the instance interface. It is therefore possible for external users of an object to change its state, when the actual objective was to hide state information. However, although inheritance has some drawbacks, it is still a very powerful mechanism [Wegner89]. Thomas [Thomas89] claims that it is the distinguishing feature which separates object oriented programming environments from other programming systems and, in all probability, without it, object oriented programming would not have gained such widespread popularity.

2.5 *Communication*

Message Passing

As has been shown, the potential behaviour associated with objects is specified by their protocol, a set of message selectors which are symbol names associated with operational code. Since the only way to access or change the state of an object is via its specified behaviour, initiating that behaviour is accomplished by sending an object a message. The effect of the message passing paradigm is to shift control from the operators to the operands (objects). For example, consider the simple expression 2 + 4. Conventional programming languages would interpret this as: the operator (function) + takes two parameters, the operands 2 and 4, and returns a result (6). In object oriented systems, that computation would be viewed as: send to the object 2 the message + with a parameter, the object 4. Conceptually, as processing activity takes place 'inside' objects themselves by executing methods [Ingalls78, Rentsch82], this example demonstrates how the object 2 is in control of the computation, not the operator (message selector) +. 'Outside' an object, the only thing which happens is the arrival of messages, delivering some request. Message senders are not concerned with how a request is carried out (whether by procedure call, a data packet sent on a distributed computer network, or even a message sent to another object [Thomas89]), as long as the requested behaviour is carried out. Generally, this conceptual shift of control highlights how: "Function calls specify not what should be accomplished but how. The function name identifies specific code to be executed. Messages, by contrast, specify what you want an object to do and leave it up to the object to decide how" [Ledbetter85]. This distinction between specifying what should be done rather than how it should be done is the essence of the message passing paradigm.

Message sending is a form of communication. However, as Wolczko [Wolczko88] points out: "Communication in object oriented languages is one of the most contentious issues in the object oriented programming community". He points to three different models of computation. In the Actor model [Agha86, Hewitt77], objects are autonomous processes and communication via message passing is completely asynchronous. In the POOL model [America86], active objects send messages between each other in an Ada style rendezvous. Finally, in the Smalltalk-80 model, message passing is completely synchronous; an object sends a message to another object and then has to wait for a reply [McCullough87]. This last model will be used for the rest of this discussion.

When an object partakes in processing activity by executing a method, there is often a need to execute some other method resident in that object. In other words, a message request needs to be sent to the object again, except that the source of the request is the object itself. Most object oriented programming systems provide a mechanism whereby objects can refer to themselves during processing, usually via a variable called `self`. However, if an object's class has overriden a specific method, but needs to execute an original definition higher up in the hierarchy, sending a message to `self` will not have the desired effect.

Many systems provide an additional mechanism to `self` called `super` to overcome this problem. Sending messages to `super` causes methods higher up the hierarchy to be accessed. The `super` mechanism thus allows inherited methods to be accessed in new methods with the same name. However, with the provision of `self`, `super` and inheritance, it is sometimes difficult to determine which methods are being executed in response to a message send, since `self` always refers to the same object during the execution of a method. This scanning up and down the class hierarchy searching for methods due to `self` and `super` is known as the 'Yo-yo' effect [Taenzer89]; see Figure 2.4. Searching the class hierarchy for a method also involves the act of *binding*, which "... is the process of making the connection between a name and the object it refers to" [Cook86].

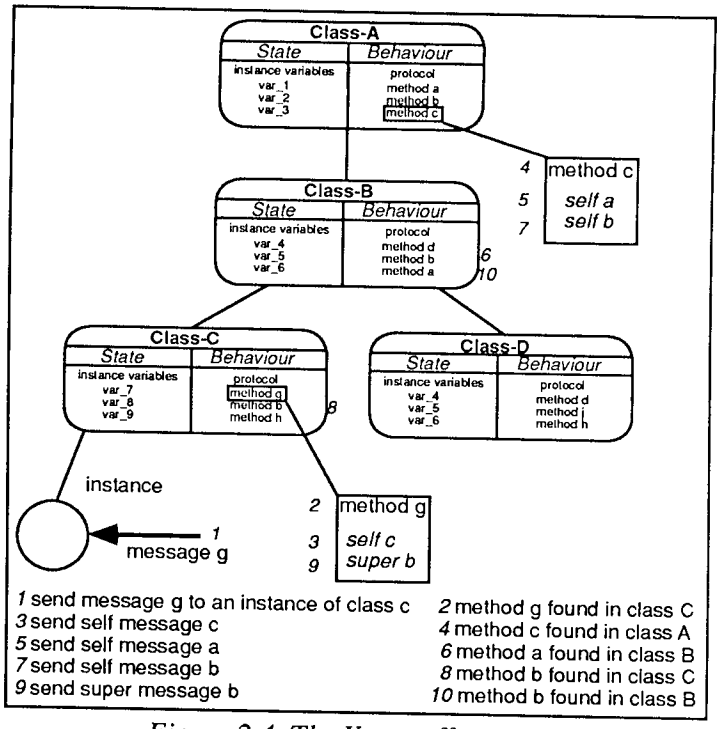


Figure 2.4. The Yo-yo effect.

Binding and Polymorphism

A binding mechanism has to consider four components: a name, a value, a type and finally whether the value bound to the name is permanent or not [Morrison88]. The following two pieces of Pascal (Figure 2.5) highlight these four aspects of binding. The only other consideration concerning the binding mechanism is when it takes place. Binding can either take place before a program is run or during running. The former is called *static binding* and is carried out by a language processor. The latter, known as *dynamic binding*, is carried out by the

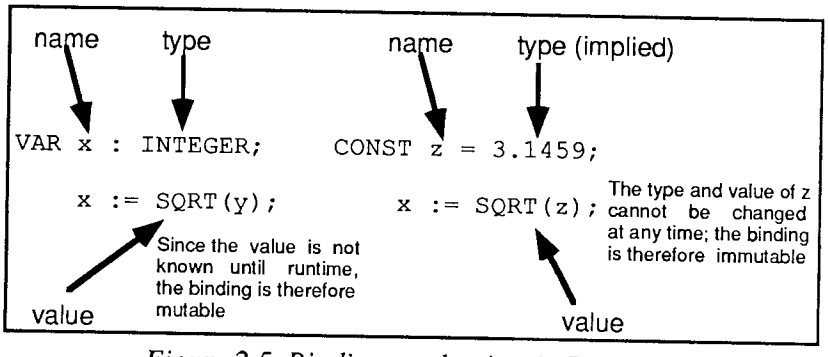


Figure 2.5. Binding mechanism in Pascal.

run-time environment in which the program resides. The advantage of static binding is that a considerable amount of consistency checking can be carried out at compile time. This identifies a large number of potential programming errors, leading to increased confidence in the correctness of eventual running programs [Atkinson88]. Also, statically bound programs can (usually) be optimised by reducing the amount of run time code, since there is no need to store type information with the data itself.

Although static binding produces more 'secure' programs, it can sometimes be too restrictive. To demonstrate this, the following is a piece of Pascal code (Figure 2.6) in which the `draw` procedure can in effect accept any shape type via a variant record:

```
program poly;
  type
    shapetag = (circle, square, triangle, star);
    polygon = record
      case shape : shapetag of
        circle   : (radius: integer;
                    position: point);
        square   : (origin: point;
                    length: integer);
        triangle : (top: point;
                    left: point;
                    right: point);
        star     : (points: array[1..5] of point)
      end;
  procedure draw (someshape: polygon);
  begin
    case someshape.shape of
      circle   : drawcircle(someshape);
      square   : drawsquare(someshape);
      triangle : drawtriangle(someshape);
      star     : drawstar(someshape)
    end
  end;
```

Figure 2.6. Simulated Polymorphism in Pascal.

Each time a new shape needs to be introduced to the program, types `shapetag`, `polygon` and procedure `draw` will have to be modified and recompilation of the application will be necessary. This approach to simulating polymorphism becomes increasingly more unmanageable as the size of the application code expands.

Polymorphism "is desirable because it enables us to write extremely general-purpose programs in a transparent manner — the bare algorithm and no frills" [Harland84]. Polymorphism means that the same message sent to different objects can give different results. As Morrison [Morrison87] states: "Polymorphism is a mechanism whereby we can abstract over type" and enables programmers to produce generalised software components. Usually, ad hoc polymorphic operations in conventional languages are provided as overloaded operations (e.g. `write`, `read`, + in Pascal). Hybrid languages such as Object Pascal with its attendent class hierarchy MacApp, provide a limited form of polymorphism as illustrated overleaf. The same procedure names can be used in different objects, so providing polymorphic operations. In the example, Figure 2.7, each shape knows how to draw itself by responding to the `draw` message and invoking the `draw` procedure:

```
program poly;
  type
    shape    = object(TObject)
                 procedure draw;
               end;
    circle   = object(shape)
                 radius   : integer;
                 position : point;
                 procedure draw; override;
               end;
    square   = object(shape)
                 origin : point;
                 length : integer;
                 procedure draw; override;
               end;
    triangle = object(shape)
                 top   : point;
                 left  : point;
                 right : point;
                 procedure draw; override;
               end;
```

Figure 2.7. Object Pascal and Polymorphism.

However, if another class, say `star`, was introduced that was not a subclass of `shape`, but included a `draw` procedure, the following code extract would fail at run-time if passed an instance of class `star`.

```
procedure highlight(someshape: TObject);
  begin
    . . . .
    shape(someshape).draw; {type coerce someshape}
    . . . .
  end.
```

Although the procedure `highlight` can accept any object type, in order to send the `draw` message to the parameter of `highlight`, it has to be type coerced (in this case to type `shape`). This bypasses the compiler's type checking, as class `TObject`, the root of the MacApp class hierarchy, does not have a `draw` procedure. When an instance of `star` is passed to `highlight`, because it is not a subclass of `shape`, the procedure crashes at run-time. To avoid problems like this, dynamic binding is needed. Arguably, dynamic binding provides a more flexible form of (ad hoc) polymorphism in a class hierarchy than static binding[3].

When a message is sent to an object, a search is initiated in that object's class. If the specified method is not found, the search continues up the class hierarchy until the method is found or an error occurs. Once a method is found, it is effectively bound to the message selector and run. As can be seen, this model of message passing is pre-requisite in overcoming the inevitable combinatorial explosion of routine complexity in extendable polymorphic systems; message sending reduces polymorphic operations to monomorphic. With the development of this form of message passing, the burden of explicit type checking and type dispatching disappears by making the routines themselves monomorphic and embedding them within system types (i.e. classes) [Ingalls86].

[3] The language Eiffel [Meyer88], although it possesses dynamic binding uses static binding as much as possible.

Two kinds of polymorphism supported by message passing are *inclusion* and *operator* polymorphism [Cardelli85]. Inclusion polymorphism is directly attributable to the inheritance and subclassing mechanisms present in object oriented systems. This kind of polymorphism is where a function will work for all types (classes) in the system and, moreover, with the same piece of code. Hence all operators at the root of the class hierarchy can be seen as the *universal polymorphic operators*, as all objects in the system can respond to them. Operator polymorphism is a result of strong encapsulation facilitating operator overloading; a system will allow many instances of the same function name to coexist. For example, if two classes, which bear no relation to each other (one is not a subclass of the other or *vice versa*), have the same name for an operation but both behave differently, this is operator polymorphism. Operator polymorphism is demonstrated in the Object Pascal example.

2.6 Summary

Three essential components of object oriented systems have now been discussed along with their attendant attributes and characteristics:

- Composition - Objects are instances of classes and are first-class encapsulated data abstractions, having a unique identity and persisting for as long as they are needed.
- Organisation - Inheritance within a subclass hierarchy leads to a reuse of code with an attendant factoring capability.
- Communication - Processing via message sending (dynamically bound procedure calls) exploits the inherent polymorphism of object systems.

Many areas of current research have not been covered in this discussion such as alternative sharing mechanisms like *delegation* [Lieberman86, Stein87, Minsky89], concurrency within object architectures [Yonezawa87] and alternatives to class mechanisms such as *prototypes* [Lieberman86, Ungar87], as these concepts do not directly impinge on the research carried out. Although this chapter is intended to be a general description of what constitutes object oriented languages, the influence of the Smalltalk-80 model should be evident.

She's a model and she's looking good. I'd like to take her home that's understood. She plays hard to get she smiles form time to time. It only takes a camera to change her mind ...

The Model — Kraftwerk[1]

3. *JSD and 'Object Orientedness'*

3.1 Overview of JSD

Background

The development of JSD can be traced back to about 1977 [Cameron89]. Michael Jackson's paper 'Information Systems: Modelling, Sequencing and Transformations' [Jackson80] criticized the functional decomposition approach to software development and introduced the essence of JSD. The paper emphasized " ... two pairs of concerns: the separation of model from function, and the separation of design from implementation". The first reference to JSD as a system development method appeared in [Jackson81]. Full details of the method were released in two books published in 1983 [Jackson83, Cameron83]. Ideas from Jackson's previous work [Jackson75] played a major role in JSD's development, and as Jackson [*ibid*] states: "JSD has grown out of JSP[2] ". The method has since evolved by the continuing efforts of Cameron and other members of MJSL[3]. Other major works that have been published on JSD include the second addition of Cameron's 1983 monograph [Cameron89] and [Sutcliffe88].

Since its inception, JSD has been used for a variety of systems[4], all of which can be regarded as having a strong time dimension. The term 'time dimension' refers to the potential concurrency of a system and the time-ordering constraints to which system events conform. A large proportion of real world systems, both real time and data processing, are implicitly concurrent [Jackson81]. Since JSD directly addresses the issue of concurrency, its use in the development of such systems is clearly appropriate.

The JSD method covers the technical development phases of the software life-cycle, starting from an existing system requirement and proceeding through to a fully implemented system (see Figure 3.1 overleaf) [Cameron86].

---

1 From the album 'Man-Machine' by Kraftwerk, © 1978 Klingklang Music, Capitol Records Inc.

2 JSP is Jackson Structured Programming.

3 Michael Jackson Systems Ltd, now absorbed into LBMS.

4 See [Cameron89] section 7; JSD Examples.

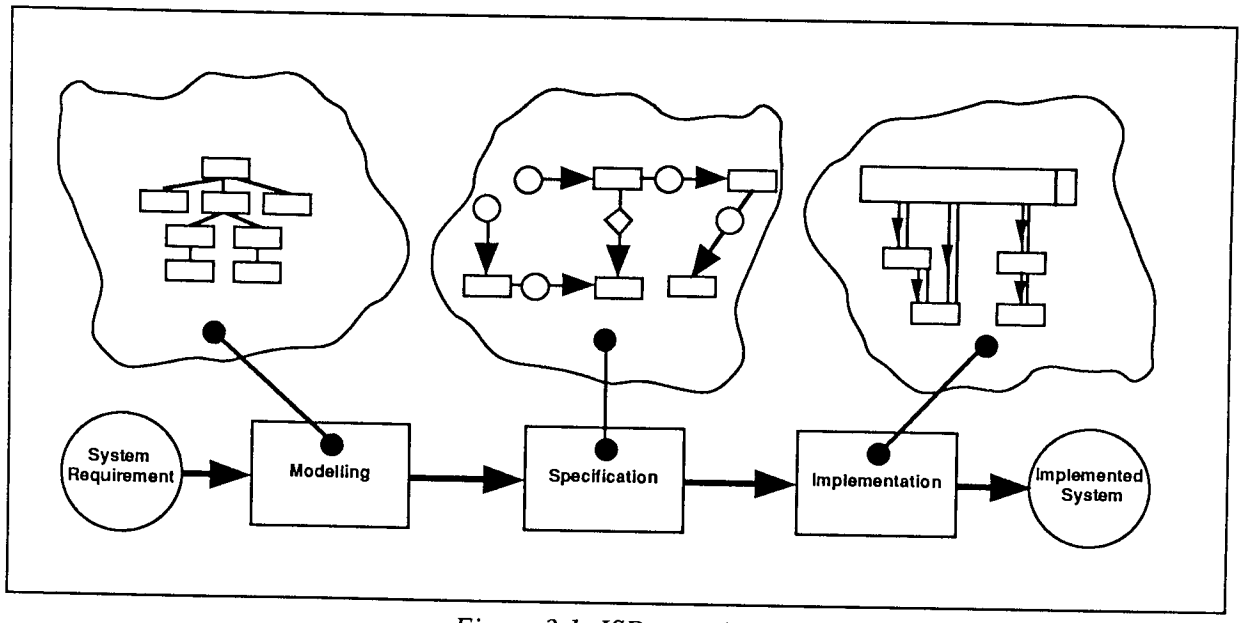


Figure 3.1. JSD overview.

The approach is systematic in that it decomposes the development task into well defined stages. In particular, there is a distinct separation between specification and implementation. The starting point in JSD is the development of a model of the real world subject matter on which an abstract system specification is to based. This approach, when compared to other development methods, gives greater insight into what is wanted by forcing a " ... developer to really understand what the system is about" [Renold88]. Functions are then built upon the model to produce the required input/output activities of the system. Finally, the specification is transformed to produce an implementation of acceptable performance. This final attribute of JSD is " ... attractive because it is not based on an arbitrary life cycle model, but simply regards the development process as a series of transformations between specifications" [Potts85].

Operational Approach

Traditionally, JSD has been associated with the operational rather than the object oriented paradigm owing to the fact that JSD specifications are (at least in principle) executable. Moreover, by explicitly modelling the problem domain in which the system is to operate, by making a clear separation between specification and implementation issues and between system modelling and function, and by adopting a transformational approach to derive system implementations, JSD firmly places itself in the operational paradigm [Agresti86]. Three other operational methods worthy of mention are Gist [Balzer82, Cohen82, Feather82], Paisley [Zave82, Zave84, Zave84a] and Me-Too [Henderson86], which is a completely formal based operational method.

Like JSD, Gist embeds an explicit model of the real-world problem domain in its specifications. Gist specifications are composed of descriptions of behaviours which " ... correspond to the observable activity in the application domain ... " The structure of a specification itself is important only insofar as it results in observable behaviour (i.e. changes to the modelled world) [Balzer82]. Gist specifications

are based on the idea of having a store of states in which activities in the outside world initiate operations in the specification to produce a new state. Unlike JSD, process behaviours in Gist specifications can be completely nondeterministic and are specified by a series of 'stimulus-response rules'. Constraints are applied to the states generated by the behaviours to reduce the non-determinism within the specification.

Paisley is similar to JSD in that its major unit of specification is the 'process'. Processes are specified in an applicative or functional (side effect free) style by supplying a 'state space' and a 'successor function' on that state space. A successor function defines the successor state for each given state on its state space. A process executes indefinitely, applying its successor function to generate new states. Each process simulates some relevant part of the problem domain under investigation and so, like JSD, explicitly captures part of the real world within its specifications.

Mee-Too is both a development method and a language. The method is divided into three steps. The first is to define a model in which abstract data objects and associated operations make up the structure and behaviour of a proposed system. The next step refines the first by expressing those abstract data objects and operations formally using abstract data types and recursion equations to produce a formal specification. Here, the Me-Too language is employed, being a mixture of Miranda [Turner86] and VDM [Jones90]. Finally, because the notation used in the second step produces a formal specification in the form of a functional program, the specification is thus executable. The third step executes the specification as a prototype to see if it produces the wanted behaviour.

Identifying a link between JSD and the object oriented paradigm is not new and several references have already made either explicit or implicit connections between the two approaches. For example, Cook [Cook86] in his discussion of object oriented programming identifies modelling the real world as the basis for building object oriented programs. He highlights this activity in the following way: "This process in itself is independent of any particular programming or design language, and a good description of the software design process in a language independent context is given in Michael Jackson's book 'System Development'"; in other words, he relates the modelling phase of JSD (see section 3.2) to the initial activities of building object oriented programs. More generally, Masiero [Masiero88] concludes that "JSD can be used as an object oriented method". A more practical expression of the link between JSD and the object oriented world is presented by Knudsen [Knudsen88] who discusses a bachelor level degree programme at Aarhaus University, Denmark. This programme teaches the JSD method as part of an object oriented programming course. Knudsen states that "the introduction to JSD is related to object oriented programming where it is emphasised that JSD in fact is very close to an object-oriented methodology" [*ibid*]. Perhaps the most significant event[5] to support a tentative connection was the invitation by OOPSLA'87 (OOPSLA is an annual international conference on object oriented programming) to M. Jackson as special guest speaker [Jackson88].

[5] In the author's opinion.

3.2 *Modelling and Object Orientedness*

Real World Representations

The first activity undertaken when constructing a JSD specification is to model the problem domain. This explicit shift of emphasis when constructing systems stands JSD apart from many other development methods: "Instead of viewing the system as primarily a device for doing something, for computing its outputs from its inputs, we should view it primarily as a model of the reality with which it is concerned" [Jackson80]. The modelling phase of JSD [Cameron88] attempts to capture that part of the problem domain in which an eventual system is to function [Cameron86]. "The JSD insistence on starting development by explicitly modelling the real world ensures that the system user's view of reality is properly embodied in the specification and, eventually, in the structure of the system itself" [Jackson83]. A JSD model is therefore an abstraction of the real world that forms the subject matter of a proposed system [Renold88]. Models scope a system by implicitly defining what functionality it can support and therefore what it cannot support; future functional enhancements to a system will be possible only if the model has captured that part of reality to which the new functions relate.

Conceptually, the modelling phase of JSD reflects the aims of the object oriented paradigm in that it attempts to simulate real world entities as objects in software. It is to this first stage of JSD, the modelling phase, that the term 'object oriented' can be applied most appropriately. Birchenough and Cameron [Birchenough89] support this view by stating that "JSD is explicitly object-oriented only during the modelling phase". Links between modelling (in the JSD sense) and object oriented programming have been highlighted indirectly from within the object oriented community itself. For example, Madsen [Madsen88] considers execution of an object oriented program "as a physical model, simulating the behaviour of either a real or imaginary part of the world". Further, Booch's [Booch86] suggestion of coupling "object-oriented development with appropriate requirements and analysis methods in order to help create our model of reality", and then stating that "we have found Jackson Structured (*sic*) Development (JSD) to be a promising match" is further support for a link between JSD and the object oriented paradigm. Finally, McNeile [McNeile89] states that "JSD has much in common with object-oriented approaches to software design", but more importantly he goes on to say that "JSD combined with suitable code generation tools, can support incremental development and prototyping ...". If, in this final remark, the word 'suitable' is replaced with 'Smalltalk-80', the statement would describe, succinctly, one of the achievements of this research.

Entity and Object Formation

The representation of models (in the JSD sense) in object systems is represented by a set of message-passing objects. However, models in JSD are somewhat different as they attempt to capture the dynamics of the things they represent. For example, a customer orders some goods, receives them and if satisfied

pays for them. There is an explicit ordering constraint imposed by the problem domain itself on the activities of ordering goods, receiving goods and paying for them; you cannot receive goods before they have been ordered. JSD views the dynamic constraints of reality as an essential part of any model that is created.

The most natural way to capture the dynamics of entities in the real world is to state the time-ordering of their events [Jackson82]. This approach to modelling entities is known as *entity-life modelling* [Sanden89]. Candidate events, known as *actions*, are initially identified and grouped together according to their time-ordering in the problem domain [Cameron86]. Actions are atomic in that they cannot be decomposed into sub-actions; furthermore, actions are considered to happen instantaneously. Each action has associated attributes describing its characteristics, e.g the action `lend` in a library system would have attributes such as `book-name`, `ISBN`, `lender-id` and `when`, action attributes are the basis on which an entity's own attributes are formed. A time-ordered set of actions which some real world entity suffers or performs — called its *life history* — represents all possible (valid) orderings of those real world actions. Life histories are described using three basic components: sequence, iteration and selection (see

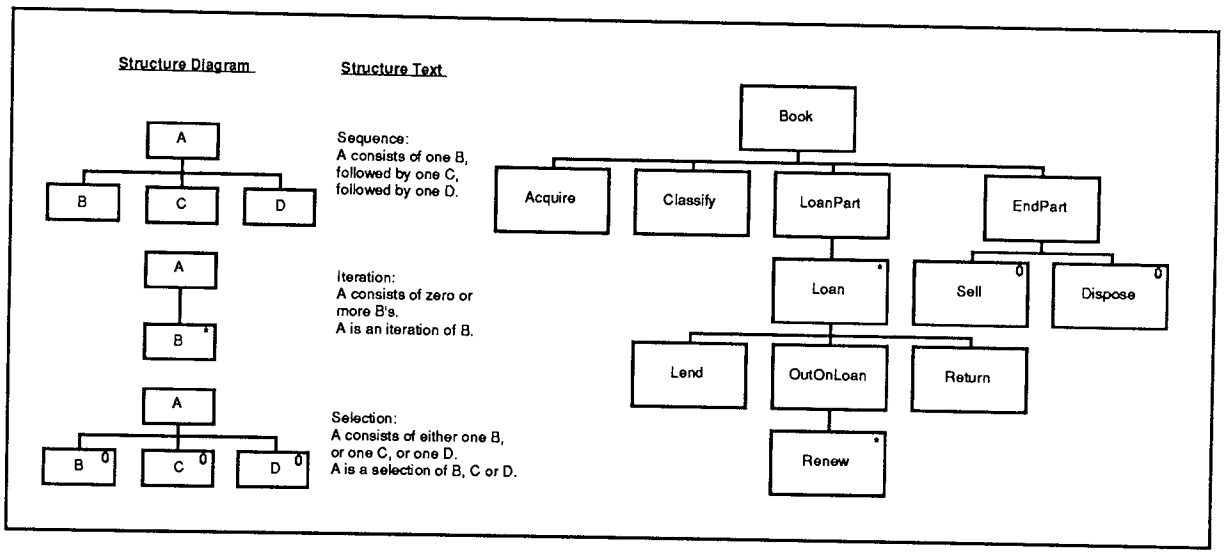


Figure 3.2. An entity-life history diagram.

Figure 3.2).

The identification of actions in the problem domain is the first activity an analyst undertakes when developing a JSD specification. In developing an object oriented program, the approach adopted is quite the reverse, as the first task is the identification of objects themselves and then their associated behaviour and the relationships between them [Shlaer89]. Experience has suggested that it is easier for analysts to identify events in the real world rather than entities, and that users find it is easier to describe how they see the real world in terms of what goes on in it, not by what entities/objects reside there [Birchenough89]. However, entity formation is still one of the most difficult parts of the JSD modelling process as it is not always clear to which potential entity an action belongs. It is possible that candidate actions do not belong to any entity as they lie outside the *model boundary*. When an action is associated with more than one entity, it is known as a *common action*. For example, in a library system, the action `lend` is common to the entity `Member` (a member is lent a book) and also to the entity `Book` (the book is lent to a member) [Cameron88]. Common actions are similar to overloaded messages (i.e. messages with the same name)

appearing in different classes and therefore exhibit a kind of polymorphism.

Great stress is placed in JSD upon the time-ordering of actions in entities representing objects of interest in the real world. This emphasis on time-ordering highlights one of the many differences between object oriented design and JSD. An object in an object oriented system has no time dimension, and no external order imposed on method invocation. A method within an object can be activated at any time, and any number of times. In JSD, the opposite is true in the sense that there is a strict ordering of actions within an entity life history — the associated behaviour of each action can only be executed when the process reaches a specific state. The principal difference, therefore, is that there exists in JSD a prerequisite to order external real world actions to formulate process entities [Jackson83], whereas in object systems there is not.

The lack of constraint imposition in the order of method invocation in class specifications is seen as a distinct advantage by some. Meyer [Meyer88] states that "it is often a mistake to freeze this order too early, as sequencing of actions is one of the aspects of system architecture that tends to change most often during development and evolution"; he goes on to say that "the facilities of a module are there for other modules to use in the order they desire". Meyer quite rightly highlights lack of time-ordering in object systems when stating that "this refusal to concentrate too early on sequential constraints is, in my view, one of the key differences between object-oriented design and ... JSD, Jackson's System Design (*sic*) method". However, there are three points to Meyer's argument in need of comment. The first is that by stressing the ordering constraint, the abstract specification will, more accurately, reflect the activities of the real world. The order of these actions does *not* change very often, as he asserts, compared to functional requirements, which is the whole point of the modelling phase of JSD — to capture, as accurately as possible, the stable features of the problem domain. Secondly, sequencing of actions is not one of the aspects of system architecture in JSD, but of the architecture of processes. Process intercommunication is the basis of system architecture in JSD specifications. Thirdly, Meyer appears to confuse the fundamental differences between JSD specifications and their potential implementations. A JSD specification is an *abstract* specification and, as such, can have many different implementations.

3.3 Specifications and Object Orientedness

Processes

A natural way of representing time-ordering of events in software systems is by using sequential processes [Jackson84]. Thus a real world model in JSD is realised by a set of long running sequential processes, each expressing the possible time-ordering of certain external real world events. Moreover, entity-life histories which are realised as sequential processes are descriptions of process classes [Jackson80]. For example, the life history of a library book described in Figure 3.2 is a description of all possible books in a library; thus, for every single book, there will be a separate sequential process modelling its life history. These instances of a process class realise entity attributes in *state vectors* — the persistent and encapsulated data about which a system can provide information via system functions. Basically, a state vector which is being continually updated by the owning process, keeps a historical

record of what the process, and hence corresponding real world entity, has done.

Instances of a process class within a network appears to parallel the class/instance mechanism employed in the object oriented paradigm. However, JSD process classes, are not templates for creating instances in the object oriented sense; the semantics of networks is such that processes are neither created or destroyed. On the other hand, the class/instance mechanism employed in the object oriented domain specifically caters for instance creation. Class instances (objects) are dynamic entities in the sense that they have a beginning and an end whereas JSD process instances do not.

Making a stronger connection between process instances and class instances can be achieved by observing one of the constraints dictated by JSD specifications — all process instances must be distinguishable from each other. Distinguishing between instances of a process class in JSD and instances of a class in object systems is very similar. Individual states are identical in structure (instances of a process class have the same set of entity attributes, and instances of a class have the same instance variables) but contain different values. Each process instance in JSD has a user-supplied unique identifier as part of its state vector, which parallels the property of identity possessed by objects though this is usually supplied as part of the object oriented runtime environment. However, there are moves to try to alter this understanding of process instance identity by making the property of identity 'external' to the process instance itself (i.e. not part of a process's state vector) [Birchenough89a]. This would then exactly parallel object identity found in environments such as Smalltalk-80.

Actions in the problem domain generate messages which are read by the system's model processes. These processes then (partially) execute and thus synchronise themselves with the dynamics of the outside real world (albeit always inevitably lagging behind [Jackson83]). It is interesting to note that the associated behavioural representations of these actions in model processes are the only components which can change the state of such processes. This behavioural activity is similar to the way object oriented systems operate — methods are the only components which can change the state of an object. The reason this similarity exists is that both support data encapsulation — processes encapsulate their state vectors and objects encapsulate their instance variables. However, the degree to which encapsulation is enforced is different. Although a state vector can be changed only by a process's associated action behaviour, the state vector is in a sense not fully encapsulated since its inspection takes place without the owning process being involved (see later).

A further property which system states have in common in the two paradigms is persistence. In object oriented systems, it is an implicit attribute of an object: "an object is persistent if it 'dies at the right time'; persistence is an observation about the lifetime of an object, namely that it exists for precisely as long as intended, and then disappears" [Low88]. In many cases, objects can last longer than the programs which use them [Atkinson87]. In JSD, state vectors of model processes persist because these processes represent the entire life spans of particular entities. State vectors become unusable only when the system itself is of no more use, not when their associated processes come to the end of their lives; the reason for this is that data might be needed for historical reasons and the end of a process's life is still a valid state[6]. This position is consistent with the fact that processes themselves are never considered destroyed. Given

[6] I am indebted to Mary Carver of LBMS for pointing this out to me.

this required persistence of process states (often realised in conventional JSD implementations in a database), it therefore seems reasonable to assume that an object oriented system with its potential for persistent programming [Atkinson87] is well suited in this respect to implementing JSD specifications.

Network Formation and Communication

Model processes provide only an abstract simulation of the real world subject matter under investigation. In the second phase of JSD, incremental development is the key, as new processes are gradually added to the model processes to specify what the system is to do; a network of communicating processes is thus created (see Figure 3.3). The new processes added are known as *function* processes as they realise the functional requirements of the system.

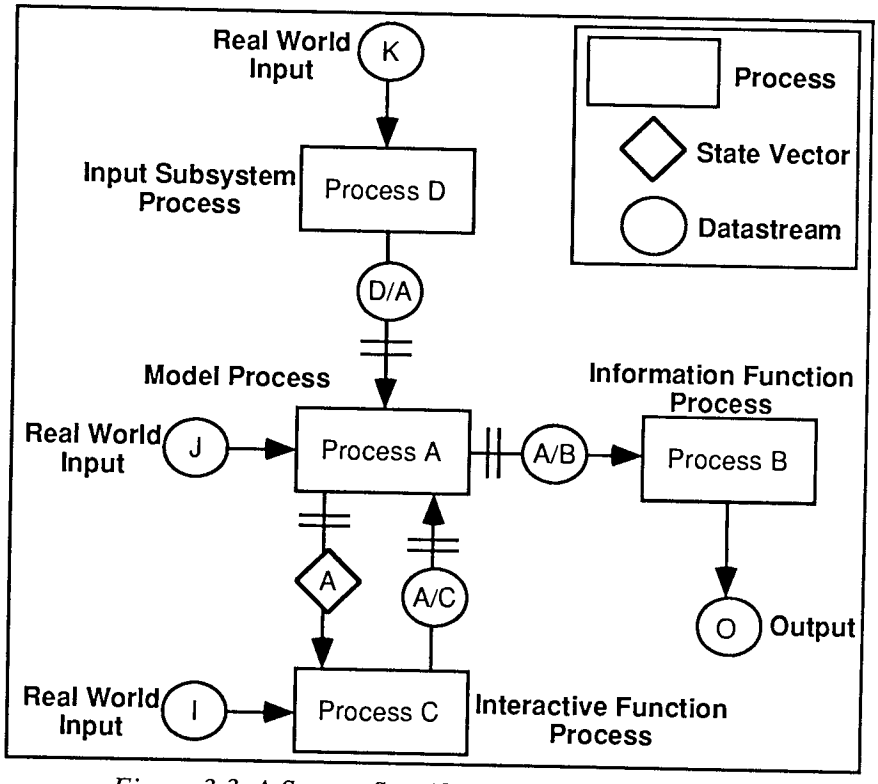


Figure 3.3. A System Specification Diagram (SSD).

Function processes are connected to model processes producing either system outputs (e.g. the circle labelled O in Figure 3.3) or additional inputs to the model processes (e.g. the circle labelled A/C) representing events not explicitly available in the real world. Other processes capture data generated by real world actions and then, after error checking, pass it on to the relevant model processes (e.g. the circle labelled D/A).

Processes communicate with each other by reading from, and writing to, idealised first-in-first-out (FIFO) buffers called *datastreams* (the circles), and by inspecting each other's internal states or state vectors (e.g. the diamond labelled A). Datastream communication is asynchronous; a process is not blocked on writing a message to a datastream, but a process is blocked when reading from a datastream that is empty. A process "takes the same time to execute its text as the object in reality which it models, possibly days, months, or years" [Renold88]. This long-running attribute of processes results in their continued suspension and resumption caused by attempting to read from currently empty datastreams at various points during their execution.

Whilst only one process can write to a datastream and only one process can read from it, processes can read from and write to many different datastreams. Where a process is reading from several

datastreams, the order in which the incoming records are to be read must be specified. Two reading strategies are possible. The first is where the reading process itself specifies exactly what the order is — this is known as *fixed merge* datastream communication (e.g. datastreams A/C, B/C and D/C in Figure 3.4). The second strategy is where it does not matter what the order is and all the datastreams are effectively coalesced into one. This is *rough merging* (e.g. datastreams E/G and F/G). Where a process class has many instances, each instance will write its own datastream. Process multiplicity is shown in Figure 3.5.

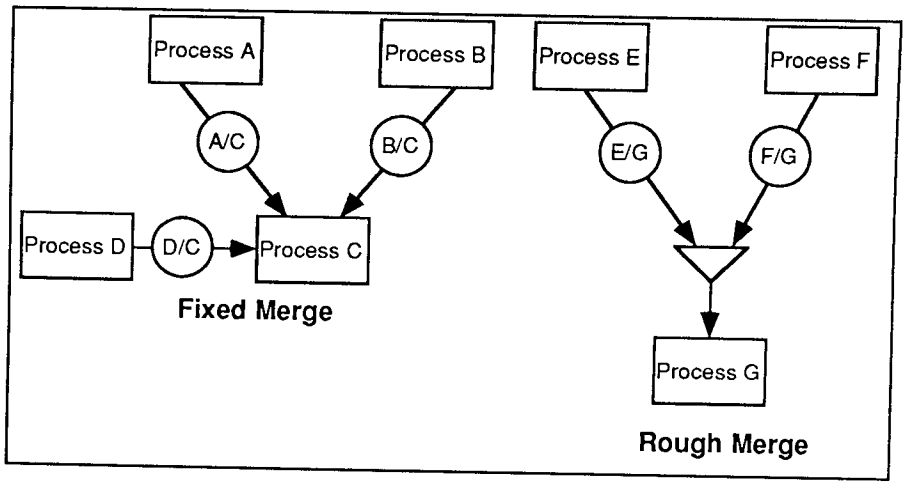


Figure 3.4. Datastream Merging.

The other form of communication used, state vector inspection, provides a read-only access mechanism permitting one process to inspect the state of another. The inspection itself does not interrupt the inspected process and the inspecting process is never blocked from carrying out the operation. The difference between state vector and datastream communication can be seen in the relationships they create between processes [Sutcliffe88]. The semantics of datastream communication creates a closely coupled producer/consumer relationship, and guarantees that every message deposited in a datastream by one process is eventually consumed by another. State vector inspection, however, does not impose any such coupling as it is completely invisible to processes being inspected. In effect, datastreams are used when data which is considered historical has to be passed around different processes; 'current state' data is handled by state vector inspection.

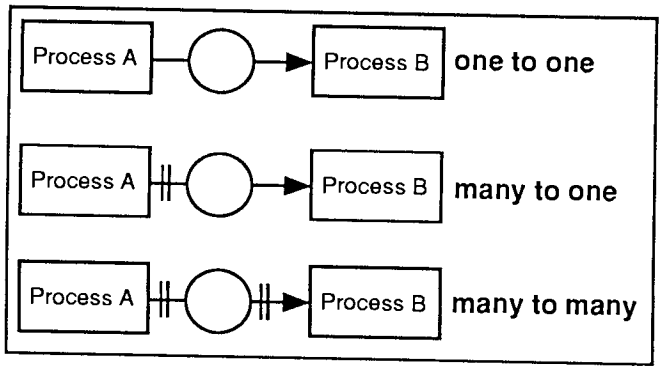


Figure 3.5. Process Multiplicity.

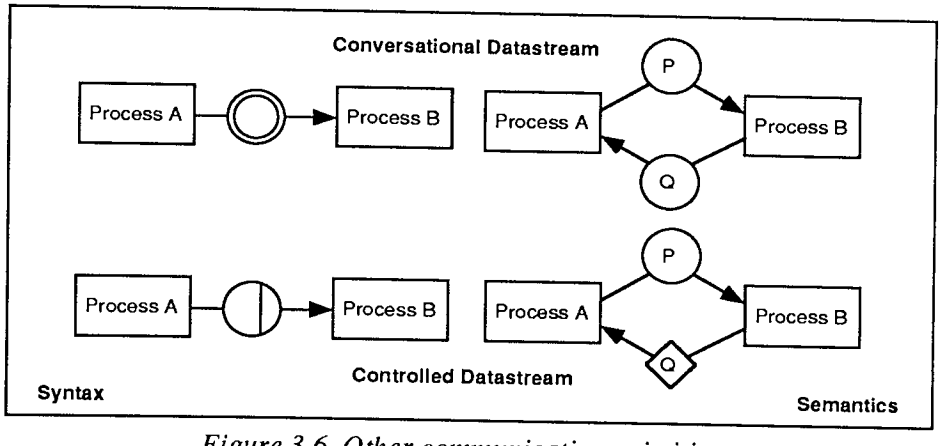


Figure 3.6. Other communication primitives.

There are two other communication primitives available at the network level, which are *controlled datastreams* and *conversational datastreams* [Renold88] (see Figure 3.6 for their notations). Controlled

datastreams ensure that a writing process can write records that the reading process is ready to accept. Effectively, this can be represented by first performing a state vector inspection and then, after ensuring the receiver process is ready to accept a record, executing a datastream write. Conversational datastreams interleave reading and writing such that once a record is written, another is written back by the receiver process. Thus, conversational datastreams couple processes more closely in the sense that only one record needs to be buffered at any one time.

It is interesting to note that both JSD and object oriented programming call their communication mechanisms 'message sending'. However, the two systems are in fact quite dissimilar. The message sending mechanism in an object oriented programming environment is an activation medium. Once a message is sent, some activity is immediately carried out by the receiver of that message; the receiver has no option but to carry out the message request immediately (assuming that there is an appropriate method for the message selector). The important characteristic of the object oriented programming message mechanism is that it can be viewed as synchronous [Wolczko88]. Objects are blocked on a message send, since they have to wait for the receiver of the message to respond. In JSD specifications, the message system employed in datastream connection is asynchronous as already discussed. Thus, processes writing records to a datastream, which can be regarded as an information passing mechanism, are never blocked. This asynchronous communication does not generally exist in the object oriented paradigm.

As regards state vector inspection, the semantics of this mechanism does not exist in the object oriented paradigm either. Because of the encapsulation mechanism employed, the only proper way to observe the state of some object is for it to provide messages which returns the value of one or more of its instance variables.

3.4 Implementation and Object Orientedness

JSD Transformations — An Overview

Whilst the semantics of JSD specifications bear some resemblance to the object oriented paradigm, the final step in the JSD method, the implementation phase, is considerably further removed. JSD uses a system specification directly to generate the desired implementation. This transformational approach ensures that the integrity of the specified system does not become corrupted during the implementation phase [Jackson83]. Generally, object oriented programs are not produced transformationally but rely completely on the ingenuity of software engineers to create them. There are a few exceptions to this. For example, one system called Producer [Cox87] translates Smalltalk-80 code into Objective-C [Cox86]. Although the system requires much programmer intervention, it does at least provide a transformational path from an interpreted, dynamically bound language (Smalltalk-80) to Objective-C and then to C.

Three major transformation techniques used in JSD are *inversion*, *state vector separation* and *dismemberment*. The following is a general overview of these three transformation techniques, further detail can be found in the chapters to follow. Inversion, which is typically applied in conjunction with

state vector separation, in its basic form converts an asynchronous producer-consumer process pair into a routine and reentrant subroutine that are behaviourally equivalent to two coroutines [Storer88, Sanden89]. When the reentrant subroutine is invoked, it resumes execution from where it was last suspended. Partial execution takes place, updating internal states, until the reentrant subroutine again suspends itself. The reentrant subroutine's suspend points are associated with the original read (or write) operations on the associated datastream, which the inversion transformation removes. The resulting implementation architecture which this transformation produces when applied over a whole (sub)network, is a hierarchy of reentrant subroutines — see Figure 3.7 which is one possible implementation of the network in Figure 3.3.

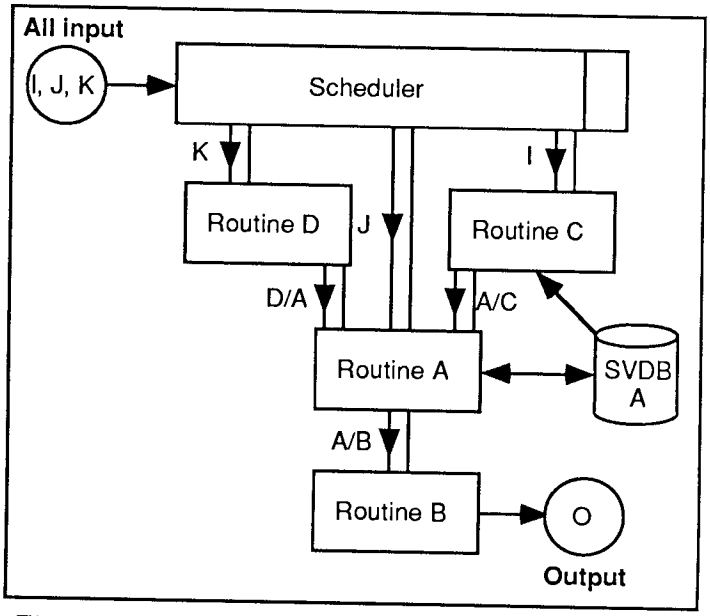


Figure 3.7. A System Implementation Diagram (SID).

JSD specifications often contain many instances of a process class, all of which are executing concurrently. These many instances can be implemented by having one copy of the process text and many copies of the state vectors. Each process is implemented by separating its state vector from its sequential procedure and storing those state vectors in a state vector database Thus it is possible, using state vector separation and inversion, to implement an entire specification on a single machine if desired. A more detailed explanation of state vector separation can be found in Chapter 6.

Dismemberment creates several new process structures from a single process structure. The execution of each newly derived structure represents a partial execution of the original process. For example, if instead of having a single piece of process text representing a model process, each action in the model is represented as a single routine, then the process would be regarded as being dismembered by state, known as *process state dismemberment.* By breaking up the structure of a process into individual modules, it is possible to schedule separately each of these modules when the system state needs to be updated, rather than executing a single reentrant routine. Some dismemberment might be desirable, or even necessary in a transaction processing environment say.

JSD Implementations and Object Oriented Architectures

One tentative connection which can be made between the two domains at this level is that the architectural structure of both is similar, i.e. both employ hierarchies. However, there the similarity ends. JSD implementation hierarchies contain stateless reentrant routines whereas in object architectures, the hierarchies are composed of classes. Further, the inheritance mechanism, which is one of the distinguishing characteristics of the object oriented paradigm, increases this gulf between the two domains as inheritance is not exploited in JSD implementation.

Another implementation linkage relates to the realisation of *roles*. Although the concept of roles falls strictly in the modelling phase of JSD, it is pertinent to discuss them here to indicate how the inheritance mechanism could be exploited to implement them. The domain of JSD specifications does not possess any inheritance property, and it would therefore seem appropriate to regard inheritance as an implementation tool [Cox86]. A role in JSD is a process which shares part of its state with another process (usually this is only the process identifier). Roles enable separate strands of concurrent activity within one entity to be modelled. For example, in a publishing bureau, high quality image setters can be viewed as having three roles. The first role models their behaviour as an asset to the company, and so activities such as their use (they should be used as much as possible to gain the best financial return) and payment of rent, etc., would be included in the role model. A second and more obvious role is the operation of the image setter itself — activities such as sending text to be rendered, indicating the number of copies required, etc., would be included in this role. Finally there could be a servicing role in which the aspects of loading more film into the machine, cleaning the roller mechanics and general servicing activities would be represented. However, the important point is that all these concurrent roles relate to the same kind of entity, namely image setters. It would therefore appear logical to organise this topology into a kind of inheritance hierarchy. In this scheme, there would be an abstract class representing the entity in which reside the common state variables amongst the roles. Each role would then be a subclass of this abstract class, all sharing the common state variables via their superclass.

Finally, another connection between the two domains involves the way object oriented systems are actually implemented. Although this connection is comparing a physical mechanism with a conceptual operation it is still worth mentioning. Most implementations of object oriented systems are based on reference semantics — manipulation of pointers. Each time a new instance of a class is created, only the instance variables are copied; the methods always reside in the class itself. A pointer to an object's parent class is maintained so that when a message is sent to an object, it is not the object which is searched to find the relevant method but the object's class instead. This physical implementation facet of object oriented languages can be seen in JSD, where behaviour is realised as stateless reentrant update procedures, which are themselves interfaced via some accessing mechanism to a database in which state vectors of the system are stored. Each stateless reentrant procedure assumes the identity of a process while executing. The result of applying state vector separation to a JSD specification resembles very closely the mechanics employed in the physical organisation of object oriented programs — the equivalent of state vector separation comes for free with the latter.

Summary

This chapter has sought to identify the points of congruence between JSD and the object oriented paradigm. The table in Figure 3.8 overleaf presents some of the features which can be found in both domains. The first phase of JSD, the modelling phase, is probably the closest to the object oriented paradigm. In both domains each attempts to capture a representation of the real world in order to build a stable system. There are superficial connections which have been identified at the specification phase

of JSD such as process and object identity, and the persistence of process instances and objects in object oriented environments. Finally, the implementation phase, the connection between the two domains becomes near non-existent. In conclusion, the differences between the two domains are such that JSD cannot really be categorised as object oriented. Probably the best pigeonhole in which to place JSD would be one labelled 'operational' as already discussed in this chapter. Nevertheless, identifying links has provided a useful indication of how to tackle the problem of transforming JSD specifications into object oriented languages.

Feature	JSD	OOP
Architectures	Networks (specification) Hierarchies (implementation)	Hierarchies
Communication	Message Passing (datastream) & State Vector Inspection	Message Passing
Encapsulation	Process (specification) & Reentrant Subroutine (implementation)	Object
Identity	User-supplied	Inherent part of Objects
Implementation Approach	Transformationally	Programmer Ingenuity
Instantiation	Process Class[1] / Process Instance	Class / Object
Message-Passing	Asynchronous (specification) Synchronous (implementation)	Synchronous
Persistence	Processes	Objects
Polymorphism	Common Actions	Operator Overloading & Inclusion
Real World Representation	Entities Model Processes	Message Passing Objects
Re-use	Possible at the Model/Network level	Automatically via Inheritance
Separating State from Behaviour	At Implementation	Cannot be done[2]
System State Change	Actions[3]	Methods
System State Representation	State Vectors (specification) Database Records (implementation)	Instance Variables
Temporal Constraints	Time Ordering of Actions	—
Unit of Modularity	Process (specification) & Reentrant Subroutine (implementation)	Class

Notes. Implementation refers to a procedural implementation via inversion and state vector separation
1. Process Classes are not templates in the object oriented sense
2. Is realised in the implementation of object oriented languages
3. For Model Processes

Figure 3.8. Overview of the features found in the two domains.

The growth of the use of transformation theory, as applied first to relativity and later to the quantum theory, is the essence of the new method in theoretical physics.

P.A.M. Dirac

4 *Transformation by Context Manipulation*

4.1 Transformations — An Overview

Transformational programming is primarily a method for constructing programs from (formal) specifications by the successive application of transformational rules [Partsch83]. However, most transformational systems are used for improving program efficiency [Balzer81], where the term 'efficiency' covers both speed of program execution and program size. One of the prerequisites of a transformational system is that it preserves the semantics of the specification or program it transforms i.e., the transformed program is behaviourally identical to its untransformed source. Thus, although transformations may alter internal structure or change some of the constructs used, transformed programs are behaviourally identical to their sources but (usually) occupy less memory and run faster[1]. Possibly one of the main disadvantages of a transformational system is that in order to increase the efficiency of programs it normally reduces their clarity. A good example of this is a compiler. Compilation is a transformation converting source code into a machine interpretable form. The efficiency of executing the machine code is greater than interpreting the source code, but to a human reader, the machine code is practically unintelligible.

In order for mechanical conversion to be undertaken, another prerequisite of a transformational system is that the source specification must be of a semi-formal nature. In fact, the *raison d'être* of transformational systems is that program construction can, to a certain extent, be carried out by computers themselves. Most transformational systems are usually to be found at the implementation end of the software development process, a fact highlighted in the classification of transformations by [Partsch83] into *program modification*, *program adaptation* and finally *program synthesis*.

Program modification systems account for most of the transformational systems in existence as the role of compilation is placed in this category. In general, program modification systems, as their name implies, simply modify some aspect of a program in a predefined manner by replacing data and control structures with (semantically similar) alternatives. Usually, predefined modifications of this type are found in compilers and are in effect optimisations, used to improve program efficiency.

Program adaptation systems are used for converting one program written in some language to (usually) the same language but on a different hardware platform or environment. One good example of a program adaptation system is that used by the developers of the Smalltalk-80 virtual machine (see

[1] One way of confirming this behavioural identity is by observing if the mappings of inputs to outputs of a program are preserved after it has been transformed.

section 4.2). Here, a generic C program representing the virtual machine is adapted for differing hardware platforms such as the Motorola 680x0 family of processors, Sun SPARC, MIPS RS2000, Intel 80386 and others [Deutsch89]. All implementations of the virtual machine are in C, but each takes into account different characteristics of the host hardware.

Finally, there is the program synthesis category into which transformations used for implementing JSD specifications fall. Program synthesis systems take, as an input, some formal representation of an abstract specification and from it generate an executable system. As already seen, in the case of JSD, the specification is a network of asynchronously communicating sequential processes.

The boundary between these three categories is not always clear cut, as many transformational systems cover more than one area. For example, the transformations used for implementing JSD specifications are primarily program synthesis but at the same time also involve program modification. In attempting to overcome this taxonomic problem, a 'two-dimensional' categorisation of transformational systems has been advanced by [Bass90] (see Figure 4.1).

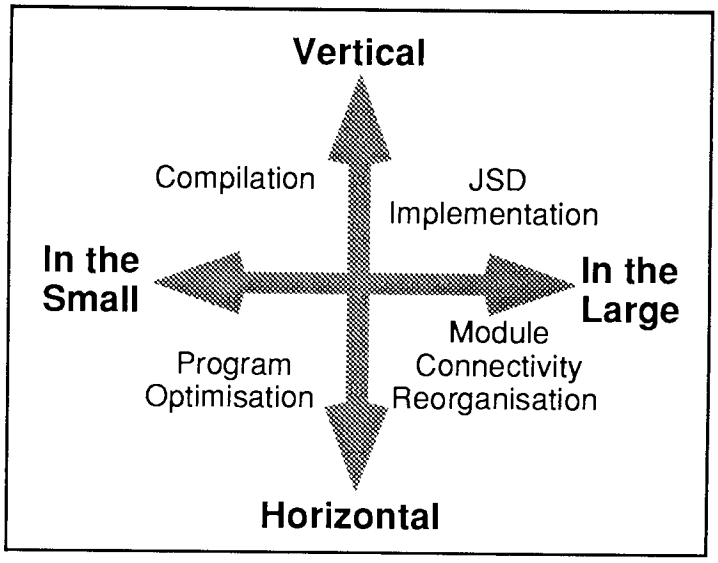


Figure 4.1. Two-dimensional transformational space.

'In the small' transformations operate at the program construct level whereas transformations 'in the large' are applied to suites of programs or modules and alter their architectural relationship. The vertical/horizontal axis covers the change in the level of abstraction. Vertical transformations such as compilation involve a significant change in abstraction level. Horizontal transformations include program adaptation systems such as the generic C program used by the implementors of the Smalltalk-80 virtual machine. Levels of abstraction are not changed, but constructs are replaced by alternatives of the same level of abstraction. Within this two-dimensional transformational space, JSD falls in the 'vertical in the large' quadrant, as the transformations alter the architecture of the specification — networks are mapped to hierarchies (in the large) and asynchronous communicating processes to conventionally invoked subroutines (vertical).

As a central goal of this research is to produce a set of general transformations for mapping JSD specifications into object oriented languages, the transformations derived will fall somewhere within this two-dimensional transformational space. Before describing such a general transformation strategy, much of the detail involves an understanding of the Smalltalk-80 environment, as this is eventually used as the target implementation platform. For completeness therefore, a brief overview of Smalltalk-80 will now be given.

4.2 *Aspects of Smalltalk-80*

Virtual Image

Smalltalk-80 has two fundamental components: a machine independent part called the *virtual image* and the machine dependent part called the *virtual machine* [Straw89]. The virtual image contains all the data structures (classes, objects) and behaviour (methods) of the system and the virtual machine interprets the methods and manages the objects in that virtual image. The virtual image, when saved to a file, can be viewed as a 'snapshot' of the contents of memory that Smalltalk-80 was occupying at the time the snapshot was taken. Since all objects are transferred from memory to file when a snapshot takes place, this gives objects in the environment persistence. However, this is only a weak form of persistence because Smalltalk-80 does not support multiple access to the objects stored, and the user is unable to save specific objects — either the entire image has to be saved or none at all.

Organisation — Objects, Classes and Metaclasses

Every entity in a virtual image is an object and as objects are always an instance of some class, a class must therefore be an instance of something else, since a class is also an object. In fact, a class is an instance of a *metaclass*. Earlier versions of Smalltalk had two types of data structure: objects, and descriptions of objects which were themselves not objects [Maes87a]. The first move away from this dual representation was in Smalltalk-76 [Ingalls78], where all classes were instances of the sole class called `Class`. This lead to all class protocols (messages which can be sent to classes themselves) being identical. Smalltalk-80 overcame this restriction by making all classes in the environment (such as `Browser`, `View`, etc.,) the sole instances of their metaclasses (`Browser class`, `View class` etc.). Metaclasses are instances of the class `Metaclass` and so the metaclass of class `Metaclass` is also an instance of the latter — the point of circularity in the environment. As each class has a reference to its superclass and to its subclasses, this enables classes to be organised into a tree structure. The root of the tree in Smalltalk-80 is the class `Object`. All messages defined here can be sent to any object in the system and can be viewed as the *universal polymorphic operators*. The class hierarchy in Smalltalk-80 supports a single-inheritance mechanism, in which methods defined in higher classes are available for use in lower subclasses. Unlike other object oriented languages (such as Object Pascal), a programmer does not have to specify whether a new method being defined overrides one in a superclass — the run time environment provides this facility automatically.

Virtual Machine

Methods are made up of source code statements which are instances of the class `String`. When a programmer initiates a compilation, code is passed onto the Smalltalk-80 system compiler (written in

Smalltalk-80) which translates this string into a stream of byte-codes (instances of class `SmallInteger`[2]). Streams of byte-codes such as these are eventually realised as instances of the class `CompiledMethod` and are stored in a class's method dictionary, an instance of class `MethodDictionary`. The byte-code interpreter understands 256 byte-code instructions and executes them in a stack-oriented fashion. These byte-codes fall into five distinct categories: pushes, stores, sends, returns and jumps. Figure 4.2 shows the two different representations of the `halt` method, defined in class `Object`. (Note that the byte-codes are in hex format. Since it is not possible to represent all possible instructions in 256 bytes, some byte-codes have extensions of either one or two bytes.)

```
halt
"This is a simple message to use for
inserting breakpoints during debugging."

NotifierView
   openContext: thisContext
   label: 'Halt encountered.'
   contents: thisContext shortStack
```

```
<41> pushLit: NotifierView
<89> pushThisContext:
<22> pushConstant: ''Halt encountered.''
<89> pushThisContext:
<D3> send: shortStack
<83 60> send: openContext:label:contents:
<87> pop
<78> returnSelf
```

Figure 4.2. Source and byte-code representation of the `halt` method.

As the only behaviour which can take place within methods is either assignment, sending messages to other objects, or returning objects, it might seem that 'low-level' processing activity such as arithmetic and comparing results cannot be carried out. In fact, the Smalltalk-80 system includes so-called *primitive* methods. Primitive methods are those which are not implemented in the Smalltalk-80 virtual image, but instead are implemented within the virtual machine. Such methods include: `+`, `-`, `==`, `suspend`, `resume`, etc.

Object Memory

Object memory is an additional abstraction over the store Smalltalk-80 uses. It provides an interface between the byte-code interpreter and the objects occupying the virtual image. The sole piece of data which the virtual machine uses to manipulate objects is an identifier called the *object pointer*. The object pointer, because of its uniqueness across all objects, gives objects their identity. Since objects come into being at run-time, they have to be stored in heap memory. Access to these objects is via an *object table*[3].

One primitive used extensively in Smalltalk-80 is `new` (defined in class `Behavior`), which is used to create new objects. Method `new` creates a new entry in the object table, allocates storage in heap memory and gives the created object a unique identity. As objects can be created at any time, one of the great benefits with which Smalltalk-80 provides the programmer is automatic memory management of these objects. Access to all objects is via the object table, and so the memory allocated for each object in heap memory can be moved around freely. When an object is no longer needed, the space it occupies is reclaimed back for further use. Garbage collection performs the task of explicitly releasing unwanted objects from memory, thereby reducing code size and ensuring secure termination of dead objects.

---

2 When inspected from within the environment; the virtual machine does not actually use instances of `SmallInteger`.

3 This is the implementation technique used in Parc Place's Smalltalk-80 version 2.3.

Object Communication

When an object is sent a message, the message selector is used as the key for searching a method dictionary. First, the virtual machine determines which class the receiver is an instance of (found in the object table). A search is then undertaken in the receiver class's method dictionary. If the selector is found, the compiled method associated with that selector is loaded and run, binding the selector with a specific implementation of the wanted method. However, because the environment supports inheritance, and methods can be defined in higher classes, the selector is not always found in a receiver class's method dictionary. In this situation, the virtual machine starts searching the receiver class's superclass for the selector until it finds it. When a selector is not found, even after searching all the way up to the root of the class hierarchy (class `Object`), the virtual machine sends to the original object the message `doesNotUnderstand:` with the original selector as a parameter converted to an instance of class `Message` (`doesNotUnderstand:` is implemented in class `Object` and effectively brings up the system debugger).

Blocks

A block is a special kind of object which enables a programmer to delay the execution of a piece of code until it is required. The following (see Figure 4.3) illustrates how the factorial function is implemented using blocks.

class Integer

```
fact
    ↑self = 0 ifTrue: [1]
           ifFalse:[self * (self - 1) fact]
```

Figure 4.3. Control using blocks.

In this implementation, the message `ifTrue:ifFalse:` is sent to the returned boolean object. The parameters of this keyword message are two blocks (since blocks are objects, they can be passed as parameters). As can be seen from Figure 4.4, the two implementations of `ifTrue:ifFalse:` are nearly identical. Each sends the `value` message to the appropriate parameter (bound to a block); `value` is a primitive method defined in class `BlockContext` and evaluates the block represented by the receiver. This arrangement of sending the `value` message to blocks passed as arguments of keyword messages is how most of the standard control structures (e.g. loops, selections, etc.) are implemented in the Smalltalk-80 environment.

class True

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ↑trueAlternativeBlock value
```

class False

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ↑falseAlternativeBlock value
```

Figure 4.4. Implementation of conditionals.

4.3 *Transformational Options and Smalltalk-80*

The Concurrency Question

Highly concurrent specifications present a potential problem to the implementor. A decision has to be made as to whether an implementation should maintain that concurrency (by implementation on parallel hardware) or whether transformations should be applied to produce a purely sequential system. This section will discuss the advantages and disadvantages of removing concurrency from a specification.

The inversion transformation (see [3.4]) is suitable for generating a hierarchy of routines from a network of asynchronously communicating processes. Nevertheless, inversion produces reentrant subroutines which are not of the simple call-and-return kind found in most programming languages (including object oriented). Reentrant routines have the characteristic of resuming execution from the point at which they were last suspended. The question therefore arises as to whether JSD processes with their collective concurrency can be accommodated in object oriented languages and/or whether the mechanism of inversion can be effectively represented.

Within the Smalltalk-80 environment there exists the class `Process` which can be used to represent many threads of execution, each thread executing in parallel with the others. Smalltalk-80 is not a concurrent programming system and was designed primarily to build sequential object oriented systems; hence it is not suited to building parallel or distributed programs [Foote89]. Nevertheless, a relatively straightforward mapping could be envisaged if concurrency were to be maintained in implementation. By making an abstract class `JSD`, say, a subclass of class `Process`, all process classes in a JSD specification could then be mapped onto subclasses of `JSD`. Datastreams in the network could be mapped directly onto the class `SharedQueue`, already present in the Smalltalk-80 environment. Class `SharedQueue` provides synchronized communication of arbitrary objects between processes. An object is transmitted from one process to another by sending an instance of `SharedQueue` the message `nextPut:`. A process receives an object by sending an instance of `SharedQueue` the message `next`. If no object has been sent when a `next` message is sent, the process requesting the object will be suspended until one is sent.

Difficulties

One problem with the above scheme is state vector inspection (see [3.4]), in that access to one process's state from another process can only be via an instance of `SharedQueue`. Also, the inspected process would have to be in a state of suspension when inspected. More fundamentally, however, the essence of this research is to identify mappings into object oriented, rather than concurrent, languages. Whilst it is true that processes are themselves objects populating object memory, this does not take into consideration the large overheads incurred when using many instances of class `Process`. Hopkins [Hopkins89] highlights this fact by saying that "processes will synchronise using conventional structures, such as semaphore objects or monitor objects. Generally, such schemes will have a high process overhead, and are best suited to a moderate-to-large level of granularity in the expression of

concurrency". Moreover, due to the characteristic of (dynamic) *sparsity* [Hull84] present in JSD specifications, a large proportion of long running processes will be in a state of suspension waiting for data; using Smalltalk-80 processes would therefore appear a somewhat wasteful use of such a costly mechanism.

A further point is that although this work uses the Smalltalk-80 language and environment as the target implementation, the aim is to find a transformational route which is sufficiently general to be used with most, if not all, object oriented languages. Since many of the other available object oriented languages do not provide classes such as `Process`, `Semaphore` and `SharedQueue` found in Smalltalk-80, this again rules out the exploitation of concurrency.

Another problem is that the developer is presented with two programming paradigms: object oriented and concurrent. It is claimed in [Yokote87] that "modelling the problem in two different level modules: objects and processes ... impairs descriptivity and understandability". In fact, the use of processes in the standard Smalltalk-80 environment is very sparse, partially substantiating Yokote's comment. Again, Hopkins [*ibid*] remarks on this aspect of Smalltalk-80 when he says that "a normal Smalltalk system contains only a few processes, and typical applications using concurrency do not create more than a few tens of processes". This last statement is important, as the number of (instances of) processes in JSD specifications can be excessive and the length of time they exist is conceptually forever. Cameron [Cameron86] highlights these characteristics in the description of a library system: "In our library, books last for up to 20 years and we have over 100 000 of them. Will our operating systems and concurrent languages allow us to run 100 000 processes concurrently for 20 years?" The answer to his question is probably not, but in Smalltalk-80's case, the answer is definitely not.

Finally, from a purely practical point of view, one hurdle which would have to be tackled in a concurrent implementation is code debugging. It would be inevitable that code generated from a specification could still contain bugs. The Smalltalk-80 system debugger can only debug code with a single thread of control and so with a concurrent implementation of a specification, the debugger would not function correctly. A possible solution to this practical problem would be to suspend all active processes and then, after debugging, resume the ones which were running before suspension. This would probably involve changes to the debugger itself and other parts of the Smalltalk-80 environment. Some of this work has already been investigated by Hopkins [*ibid*]. However, it is not within the scope of this research to re-write large portions of the Smalltalk-80 environment.

In conclusion, it is clear that the approach to implementing JSD specifications in an object oriented language must be to remove the concurrency. This requires that the mechanics of inversion be realised. Inversion in procedural languages is relatively straight-forward, as will be shown next. It should be noted that although state vector separation is needed when implementing JSD specifications in object oriented languages, its realisation is quite simple and so discussion of it will be left until a later chapter.

Inversion coupled with state vector separation can be used to transform a JSD specification into a single schedulable implementation. As an example, Figure 4.5 shows a portion of some hypothetical network along with the specification of one of the processes. One implementation strategy is to apply 'read-inversion' [Ratcliff89] to all processes in the network converting all the reads (except for the very first which gets deleted) in each process to suspend points and making each process a reentrant procedure. To make sure that each inverted process in the specification gets its proper share of processor time, a special-purpose scheduler process usually has to be designed that implements any timing constraints the system must satisfy. For each datastream removed by inversion, the associated data must now be passed to the appropriate reentrant procedure via a parameter from its controlling routine; this is illustrated by the left hand column of code in Figure 4.6 where process `Example` has now become procedure `Inverted-Example`.

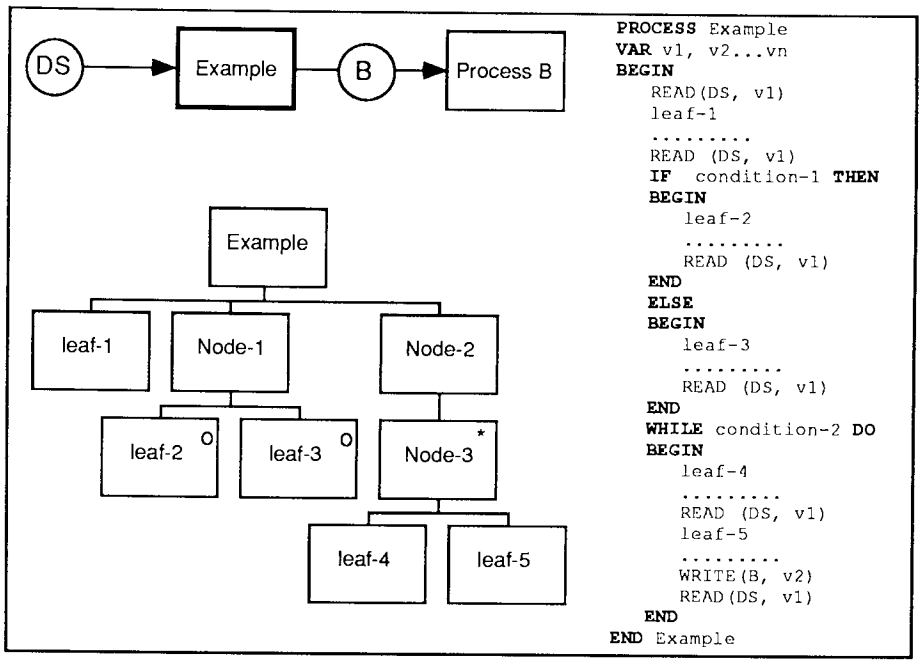


Figure 4.5. Process specification with textual representation.

The result of applying state vector separation to a reentrant procedure can be seen in Figure 4.6, in the right hand column of code. A more detailed explanation of the mechanics of state vector separation can be found in Chapter 6. Although a

```
PROCEDURE Inverted-Example(v1)
VAR  v2...vn
BEGIN
   leaf-1
   .....
   suspend
   IF condition-1 THEN
   BEGIN
      leaf-2
      .....
      suspend
   END
   ELSE
   BEGIN
      leaf-3
      .....
      suspend
   END
   WHILE condition-2 DO
   BEGIN
      leaf-4
      .....
      suspend
      leaf-5
      .....
      call Inverted-B(v2)
      suspend
   END
END Inverted-Example
```

```
PROCEDURE Inverted-Example(v1)
BEGIN
   call loadSV(example, stateVector);
   leaf-1
   .....
   suspend
   IF condition-1 THEN
   BEGIN
      leaf-2
      .....
      suspend
   END
   ELSE
   BEGIN
      leaf-3
      .....
      suspend
   END
   WHILE condition-2 DO
   BEGIN
      leaf-4
      .....
      suspend
      leaf-5
      .....
      call loadSV(procB, anSV)
      call Inverted-B(stateVector.v2, anSV)
      call saveSV(procB, anSV)
      suspend
   END
   call saveSV(example, stateVector)
END Inverted-Example
```

Figure 4.6. Application of inversion and state vector separation.

stateless reentrant procedure has been generated from a sequential process, additional vertical, in-the-small transformations are needed since most languages do not possess a suspend/resume mechanism. The transformations used convert the suspends to returns and the required resumption is catered for by placing a 'dispatcher' (a CASE statement with attendant GOTO's) at the start of the procedure and embedding labels throughout the rest of its text. Figure 4.7 (the first column of code) represents a more concrete realisation of the second column of code from Figure 4.6, (the additional state vector field selector QS keeps track of the correct resumption point of the sequential code). Also, since most block structured languages do not allow control to jump into blocks, the code is 'flattened'; the result of this can be seen in Figure 4.7, second column of code.

Inversion in hybrid object oriented languages such as C++ and Object Pascal is relatively simple; implementation would simply rely on the procedural aspects of those hybrid languages. However, in pure object oriented languages such as Smalltalk-80, it is not obvious how to successfully exploit objects and message passing to implement this mechanism. As shown above, program inversion architecturally changes a communicating sequential processes into a reentrant subroutine. However, most languages including Smalltalk-80 do not possess a coroutine facility [Haynes86, Pratt84], and so the latter has to be simulated using available language constructs. As a result, conventional realisations of inversion necessitate the extensive use of the 'goto' statement [Storer88]. Smalltalk-80, however, does not possess this construct. Nevertheless, inversion can be realised in Smalltalk-80 and the next section describes a possible approach.

```
PROCEDURE Example(someData: dsRec);
LABEL 1, 2, 3, 4, 5, 6, 999;
BEGIN
    call loadSV(example, stateVector);
    CASE stateVector.QS OF
       1: GOTO 1;
       2: GOTO 2;
       3: GOTO 3;
       4: GOTO 4;
       5: GOTO 5;
       6: GOTO 6
    END;
1:;
    leaf-1
    .........
    stateVector.QS := 2;
    GOTO 999; 2:;
    IF condition-1 THEN
    BEGIN
       leaf-2
       .........
       stateVector.QS := 3;
       GOTO 999; 3:;
    END
    ELSE
    BEGIN
       leaf-3
       .........
       stateVector.QS := 4;
       GOTO 999; 4:;
    END;
    WHILE condition-2 DO
    BEGIN
       leaf-4
       .........
       stateVector.QS := 5;
       GOTO 999; 5:;
       leaf-5
       .........
       loadSV(procB, anSV);
       call B(stateVector.v2, anSV);
       saveSV(procB, anSV;
       stateVector.QS := 6;
       GOTO 999; 6:;
    END
999:
END;
```

```
PROCEDURE Example(someData: dsRec);
LABEL 1, 2, 3, 4, 5, 6,
      100, 200, 300, 400, 999;
BEGIN
    call loadSV(example, stateVector);
    CASE stateVector.QS OF
       1: GOTO 1;
       2: GOTO 2;
       3: GOTO 3;
       4: GOTO 4;
       5: GOTO 5;
       6: GOTO 6
    END;
1:;
    leaf-1
    .........
    stateVector.QS := 2;
    GOTO 999; 2:;
    IF NOT condition-1 THEN GOTO 100;
    leaf-2
    .........
    stateVector.QS := 3;
    GOTO 999; 3:;
    GOTO 200;
100:leaf-3
    .........
    stateVector.QS := 4;
    GOTO 999; 4:;
200:;
300:IF NOT condition-2 THEN GOTO 400;
    leaf-4
    .........
    stateVector.QS := 5;
    GOTO 999; 5:;
    leaf-5
    .........
    loadSV(procB, anSV);
    call B(stateVector.v2, anSV);
    saveSV(procB, anSV;
    stateVector.QS := 6;
    GOTO 999; 6:;
    GOTO 300; 400:;
999:
END;
```

Figure 4.7. Code flattening in a Pascal like language.

4.4 *Realising Inversion in Smalltalk-80*

First-class Contexts and Reentrant Procedures

Smalltalk-80's sole use of the object/message paradigm for all programming activity gives rise to the unusual feature of making the virtual machine's runtime state (consisting of method activations and blocks) visible to the programmer as data objects [Deutsch84]. In Smalltalk-80 terminology, stack-frames are called *contexts* [Goldberg83]. Conceptually, when a message is sent to an object, a new context is created for the associated method activation by the virtual machine sending the message `sender:receiver:method:arguments:` to the class `MethodContext`. Since all processing throughout the system is accomplished by sending messages, there will be many contexts in the system at any one time. The context associated with the method currently being evaluated is called the *active* context. When the method associated with the active context evaluates a message expression, the active context is suspended and a new context is created and made active. The active context stores the context which activated it; the latter is called the active context's *sender* (which is akin to a procedure's caller in procedural languages) and is held in the instance variable `sender`. Sender contexts resume when active contexts terminate by returning.

Access to active contexts is possible via the pseudo-variable `thisContext` [Goldberg83]. Contexts accessed in `thisContext` are first-class objects and so give them the status of being *continuations* [Haynes87]. They offer a protocol allowing inspection of the program counter, stack pointer, sender, etc. and are used extensively by the system debugger. Manipulation of system contexts in Smalltalk-80 to enable the realisation of different control structures stems from the language's (partial) reflexive nature [Foote89]. Exploiting Smalltalk-80's reflexive capability by manipulating contexts facilitates the possibility of building an object which behaves like a reentrant procedure [Haynes87].

Realisation of reentrant procedures, or more accurately reentrant methods, has been achieved by developing a new abstract class called `ReEntrantObject`, a subclass of class `Object` (see Appendix A for complete list of code for class `ReEntrantObject` plus supporting code from class `ContextPart`). The two most important messages to which an instance of class `ReEntrantObject` can be sent are `suspend` and `resume`. Note, however, that the implementation of these protocols is in the virtual image, not primitively in the virtual machine as is the case for class `Process`.

Instances of class `ReEntrantObject` are used for storing the point in a method from which that method has returned. Instead of using the standard return mechanism of methods (the uparrow '↑' construct), the message `self suspend` is used instead. The reason `self suspend` is used is that when a method returns to its sender via the usual return mechanism, the instance of `MethodContext` used to run that method is lost. All methods in subclasses of `ReEntrantObject` can potentially be suspended and resumed over and above the usual send/return mechanisms offered by message passing by embedding the `self suspend` message pattern within them.

In order to explain the operational mechanics of class `ReEntrantObject`, consider the following Smalltalk-80 code overleaf (Figure 4.8). The method `controlMethod` sets a counter to zero and then

creates a new instance of class `ReEntrantExample` (a subclass of `ReEntrantObject`) which is assigned to the local variable `rObject`; this instance has the message `loop` sent to it. Once control returns back to `controlMethod` (ignoring for the moment how this is done), there follows an iteration which increments the counter per repetition; the important statement in the iteration is where the instance of `ReEntrantExample` is sent the message `resume`.

```
controlMethod
| rObject count |
count ← 0.
rObject ← ReEntrantExample new.
rObject loop.
[count < 100]
   whileTrue:
      [rObject resume.
      count ← count + 1]
```

```
loop
| count |
count ← 0.
[true]
   whileTrue:
      [count ← count + 1.
      self suspend]
```

Figure 4.8. Reentrant methods illustrating the use of class `ReEntrantObject`.

`loop` is a method defined in class `ReEntrantExample` which simply sets a counter to zero and then iterates forever; the important statement here is `self suspend` in the `whileTrue:` block. `loop` returns to `controlMethod` at the point `self suspend`; `loop` will continue execution whenever resumed from `controlMethod` immediately after `self suspend`. The implementation of `suspend` returns control but saves the context in which it appeared (in this case the context associated with the activation of `loop`). The `resume` method loads a saved context and then continues execution with the loaded context.

Implementation of `suspend` and `resume`

The above gives a general indication of what takes place when `suspend` and `resume` are used. The next two diagrams (Figures 4.9 and 4.10), illustrate the implementation details of these two mechanisms.

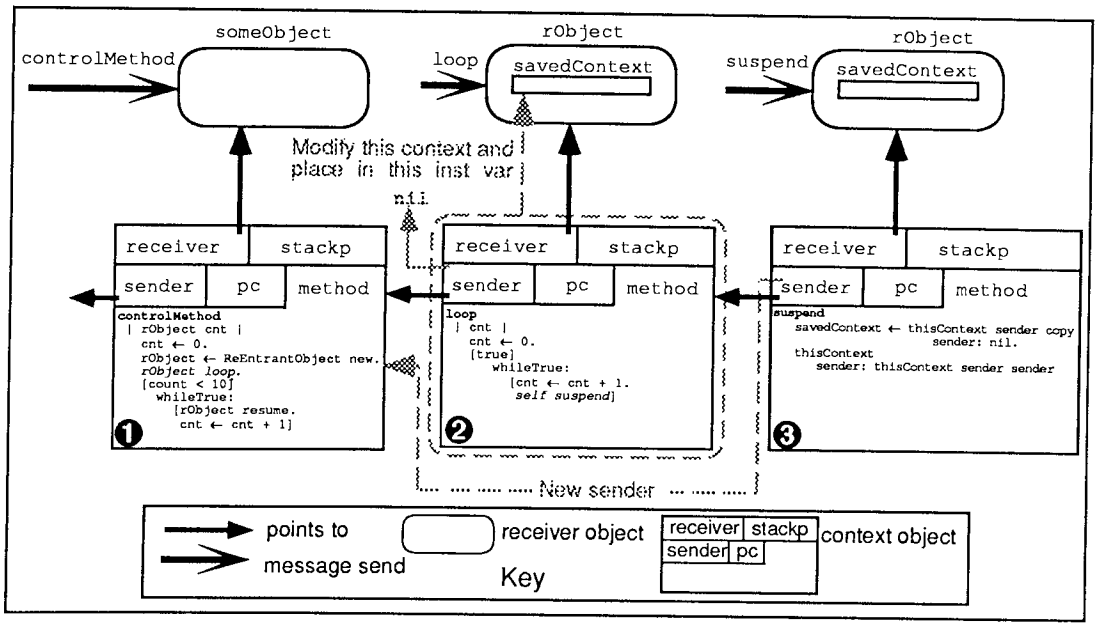


Figure 4.9. Suspend mechanism.

In Figure 4.9, a schematic overview of the flow of control for method suspension is shown, starting with the message `controlMethod` being sent to an object `someObject`, which creates a new context for `controlMethod`'s activation; this is at ❶. Within `controlMethod`, a new instance of `ReEntrantExample` is created and assigned to the variable `rObject`. `rObject` has the message `loop` sent to it, activating method `loop` (at ❷), starting the infinite iteration described earlier. Eventually, `self suspend` is evaluated in the `loop` method.

The implementation of `suspend` is shown at ❸ where the active context has its characteristics altered (i.e. the context at ❸ associated with evaluating the `suspend` method is accessed via the Smalltalk-80 pseudo-variable `thisContext`). The details are as follows:

i. The active context's sender context at ❷ (`thisContext` ***sender***) is first copied (`thisContext sender` ***copy***). The sender context is copied because the context associated with the `suspend` method (at ❸) is not the one that needs to be saved but the context which activated it (in this case the context associated with the activation of method `loop` at ❷). These contexts must be copied, because the next step breaks their sender chain.

ii. The copied context has its sender (currently pointing to the context associated with the activation of method `controlMethod` at ❶) set to `nil` (`thisContext sender copy` ***sender: nil***). This avoids saving the entire chain of contexts, when only the one is needed.

iii. The copied and altered context is now stored (***savedContext*** ← `thisContext sender copy sender: nil`) in the instance variable `savedContext`.

iv. In order for `suspend` to return not to `loop` but to `controlMethod`, the second statement in the `suspend` method achieves this by making the active context's sender (currently the context associated with the activation of method `loop` at ❷) the active context's sender's sender (the context associated with the activation of method `controlMethod`). This effectively re-links the context at ❸ to that at ❶. (Note, if the context associated with the activation of `loop` in (i) is not copied, it would not be possible to now access the current active context's sender's sender, since the sender chain has been broken in (ii)). Hence once `suspend` returns, it will return not to the context at ❷ (the context associated with the activation of `loop` which has now been copied and saved) but to the context at ❶.

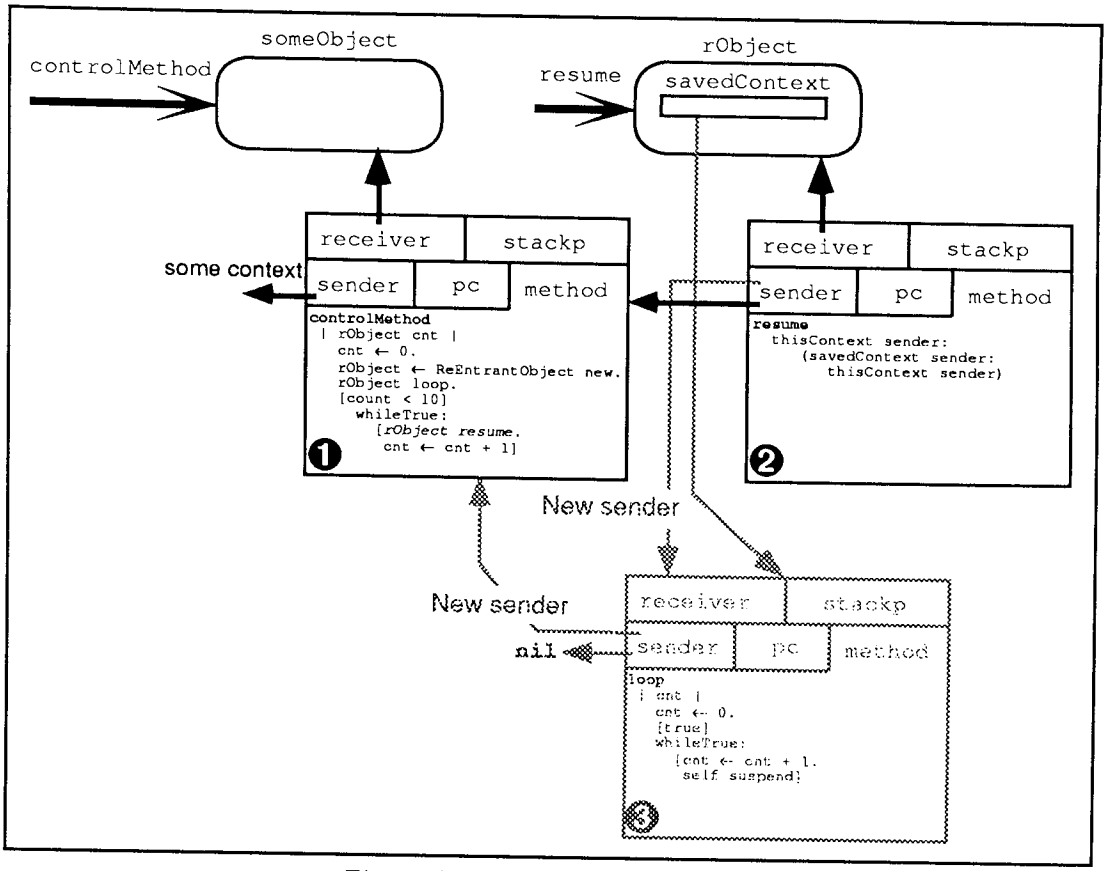


Figure 4.10. Resume mechanism.

Method resumption using `resume` is implemented as follows (see Figure 4.10). In the context associated with the activation of method `controlMethod`, the message expression `rObject resume` will be evaluated, highlighted at ❶. Sending the message `resume` to `rObject` will suspend the active context and create a new context for that message send (at ❷). The `resume` context is now active and has its characteristics altered thus:

i. The context which was saved in the instance variable `savedContext` by the suspension mechanism described above has the value of its sender (currently `nil`) made to that of the active context's sender, i.e. the context associated with the activation of `controlMethod` at ❶ (`savedContext` ***sender: thisContext sender***). This enables the saved context to return to the correct place after it has either finished or is suspended again (note that returning to a context which has no sender causes the virtual machine to crash).

ii. The value for the active context's sender (currently pointing to the context associated with `controlMethod`) is changed by making it the context in the instance variable `savedContext` (`thisContext` ***sender: (savedContext sender: thisContext sender)***). Thus, when `resume` returns, it will no longer return to the context at ❶ but to the context which was saved in the instance variable `savedContext` placed there by the `suspend` at ❸.

iii. The saved context is now re-activated. Since the context had been faithfully copied, it will continue execution from the point after the `suspend` message.

To see why context re-activation starts immediately *after* the `self suspend` message, it is necessary to understand part of the mechanics of the virtual machine during its interpretation of byte-codes. The byte-code interpreter carries out three operations which it cycles through continuously [Goldberg83]. First, the interpreter fetches the byte-code from an instance of `CompiledMethod` (held in the instance variable `method` shown in the preceding two diagrams) which is pointed to by the program counter (instance variable `pc`). Next, the interpreter increments the program counter. Finally, the function specified by the fetched byte-code is performed. When the byte-code representation of the message expression `self suspend` is evaluated, on top of the interpreter's stack will be a message receiver (in this case `self`). The interpreter will then fetch the send byte-code [Goldberg83] (which will specify the `suspend` message) and increment the program counter. Note that the state of the associated instance of `MethodContext` will now have a program counter pointing to the byte-code *after* the current send byte-code in the associated `CompiledMethod`. On performing the send byte code, the active context suspends and a new one is created. Since the interpreter has incremented the program counter automatically, on resumption, the suspended context will continue execution from the correct point in the `CompiledMethod`.

Applications of `ReEntrantObjects`

Although the mechanics of the `suspend` and `resume` methods are quite complicated (even though the number of supporting methods used to implement them is small), their use is straightforward as already demonstrated in the given example. It can now be seen that the equivalent of an inverted routine is realisable in instances of (subclasses of) `ReEntrantObject`. As an example of using `suspend` and `resume` in transforming JSD processes, the following (Figure 4.11) illustrates the realisation of the sequential structure of the hypothetical process from Figure 4.5.

```
self leaf-1.
self suspend: #leaf-1.
self leaf-2C  ifTrue:  [self leaf-2.
                        self suspend: #leaf-2].
              ifFalse: [self leaf-3.
                        self suspend: #leaf-3].
[self node-3C]
       whileTrue:
              [self leaf-4.
              self suspend: #leaf-4.
              self leaf-5.
              self suspend: #leaf-5]
```

Figure 4.11. Smalltalk-80 implementation of a hypothetical process.

In this example, an extension to the `suspend` construct has been made such that when a context returns to its sender, a parameter can be passed back (in this case indicating at what point suspension took place). Code such as that shown above could be generated quite easily by a tool from a process specification and then installed as a method in a class which was a subclass of `ReEntrantObject`.

Another application of `ReEntrantObject` is to use contexts for use as non-local exits for methods. Non-local exits are used when some error condition or exception occurs during a computation. The following example is based on that given by [Haynes86]. The function `powerFact` takes as a parameter a list of numbers and, for each number in the list, calculates the factorial of that number and multiplies the result by the original number. It does this for all the numbers in the list and multiplies all the results together. For example, `powerFact(1 2 3 4)` would be:

```
1! * 1 * 2! * 2 * 3! * 3 * 4! * 4 = 1 * 1 * 2 * 2 * 6 * 3 * 24 * 4 = 6912
```

Obviously, if a zero occurs in the list of numbers, then the result of `powerFact` is zero. There are many potential approaches to implementing the functionality of `powerFact` but for the purposes of demonstrating the power of contexts, four will be given. The first is simply to carry out the factorial computations (recursively) and multiplications, regardless of whether a zero occurs, for all the numbers in the list. The next approach is similar to this 'brute force' method but tests for zero before a recursive calculation is done; if zero is encountered, then zero is returned but, in so doing, the stack of recursive calls is unwound. The third approach attempts to overcome the wastefulness of the two previous approaches, by initially scanning the entire list for a zero; if no zero is found, then the first approach is used to calculate the result.

The fourth approach makes use of contexts, and overcomes the need to scan the list initially for a zero and carry out any calculation on stack unwind. Basically, the `powerFact` function is realised as the method `powerFact:` which is implemented recursively. At invocation, the `powerFact:` message is tagged, i.e. its context is stored for future use. `powerFact:` then recursively reads in each number from the list, but does not carry out any calculation. If no zero is detected, then on unwinding the recursive calls of itself, the calculations are performed. If, on the other hand, a zero is encountered during the reading of the list, then instead of unwinding the stack (and hence carrying out some of the calculation as in approach two), `powerFact:` immediately returns to the tagged context and processing resumes there. The implementation of `powerFact:` and all associated code is shown in Appendix B (class `ReEntrantExamples`).

Problems

The power of contexts comes from their first-class status and the fact that any sequential control structure can be built using them [Haynes86]. Other, more powerful, control mechanisms have been developed such as a backtracking system (similar to that provided by Prolog [Clocksin81]) which enables computations to have multiple solutions via the reinstantiation of contexts [LaLonde88][4].

However, the manipulation of Smalltalk-80 contexts to realise different control strategies like the `suspend` and `resume` methods does incur a small overhead (although not as great as using many instances of class `Process`). The explanation for this overhead is in the way contexts are handled in the implementation of the virtual machine [Moss87]. In order for Smalltalk-80's performance to be acceptable, the virtual machine uses some special optimisations relating to context manipulation giving up to an eight-fold increase in overall virtual machine performance [Deutsch84]. It has been shown that over 85% of all message passing within the Smalltalk-80 environment is semantically identical to the

[4] The author was unfortunately unaware of this work while developing `ReEntrantObject`.

procedure call mechanism in conventional languages — the contextual information for method activations is never accessed. Pushing and popping of stack frame information for a standard procedure call is supported by many processor instruction sets and so a large proportion of the message passing processing can exploit this. However, 15% of message passing in the environment needs access to contextual information, making the efficient implementation of method (and block) activations quite difficult — an inevitable result of making contexts first-class objects.

Instead of creating a new context object at every message send, the virtual machine creates a standard procedure activation which is pushed onto its internal stack [Miranda87, Baden84]. When a method returns, the internal activation is popped off the stack. Only when a context needs to be explicitly used does the virtual machine convert it to a real object. However, contexts which are in real object format cannot be used by the virtual machine during byte code interpretation and so have to be converted back to virtual machine format. This optimisation of having multiple representations of contexts [Deutsch84] in the virtual machine increases the performance of the system generally but becomes expensive when the virtual machine is forced continuously to convert between these representations, as is the case for the `suspend` and `resume` methods in `ReEntrantObject` already discussed. It has been reported in [Deutsch84] that this conversion exercise occupies on average about 3% of the total virtual machine execution time.

Although the `ReEntrantObject` approach works well (contrary to [Lewis90]), it is not easily generalisable to other language environments. As the thrust of this thesis is to produce JSD transformations for object oriented languages in general, it must therefore include both pure (e.g. Smalltalk-80) and hybrid (e.g. C++) object systems. Very few programming language systems offer the capability to manipulate run-time stack frames. On the other hand, some of the pure object oriented languages do not offer a 'goto' construct. This situation leads to the need for developing an implementation of inversion without using either goto's or stack-frame manipulation, and this is the subject of the next chapter.

The nineteenth œcumenical council declares that the whole substance of the bread is transformed into the body of Christ and the whole substance of wine is transformed into the blood of Christ. This transformation is called Transubstantiation.

Council of Trent. c. 1545 - 1563

5 *Followset based Transformations*

5.1 *Followsets*

Introduction

It is clear from the discussion in Chapter 4 that an alternative to a 'goto' or 'context manipulated' realisation of inversion is needed. The alternative which will be described relies on destructuring the original process and reforming it into a collection of *dismembered* components. Once this new representation of the original process has been derived, inversion and state vector separation can still be applied. As the destructuring is purely mechanical, this two-stage approach is still transformational.

The basis of the new approach is the use of *followsets*. Followsets are not a new concept but their use in the manner to be described here is new. Followsets have been applied mainly in the domain of error detection and correction in predictive parsers originating from the world of compiler theory [Aho85, Stirling85]. Basically, when a program is being compiled, for any input token, there exists a set of tokens each of whose members can follow this token. This set is the followset of the input token. Hence, for any given input token, its followset defines the next valid token set according to the syntax of the language. A token which is scanned but is not in the followset for a given context instigates the error-handling facilities of the parser.

```
process A                    process context-A
begin                        begin
   read(rec, DS2)               read(rec, DS1)
   call B                       while rec <> FIRST(A) do
   read(rec, DS2)                  read(rec, DS1)
   while condition1 do          write(rec, DS2)
   begin                        read(rec, DS1)
      call D                    while rec <> FOLLOW(B) do
      read(rec, DS2)               read(rec, DS1)
   end                          while rec = FIRST(D) do
   if condition2 then           begin
      call F                       write(rec, DS2)
   else                            read(rec, DS1)
      call G                       while rec <> FOLLOW(D) do
end                                   read(rec, DS1)
                                end
                                write(rec, DS2)
                             end
```

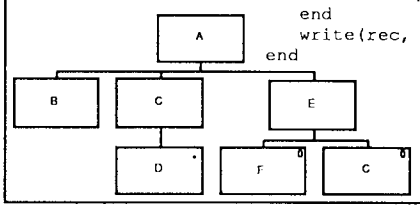


Figure 5.1. Systematic process elaboration producing a context filter.

Although originating from compiler theory, one application of followsets in the domain of JSD has already been described by Jackson [Jackson83]. Here, followsets are used for the systematic generation of general conditions associated with iterations and selections in process structures. These followsets can then be used to generate *context filters* which detect a certain class of errors in the specification's input subsystem.

A context filter is a special type of process whose sole function is to guarantee that only correct messages are passed onto model processes. Figure 5.1 gives a simple example. The specification of the

model process A is in both diagrammatic and textual form. The right-hand piece of text represents the context filter for process A. Note that embedded within the text is the application of two functions FIRST and FOLLOW to generate the followsets which are used to filter out all data records except those the model process is next expecting; these two functions are described in [5.2]. This simple use of followsets in JSD can be extended when one considers the usual representation of processes — tree structure diagrams.

JSD tree structure diagrams are a representation of regular expressions [Hughes79]. A tree structure representation of a process therefore defines all possible state sequences the process can go through. As such, a process structure diagram can be viewed as a form of finite state graph, as shown in Figure 5.2, mapping all potential states to their successor states [Put88, Zave85].

Since a *followmaplet* represents a mapping of all possible successor follow states (represented as a followset) from some initial state, it should be possible for the complete semantics of a process, as depicted in a tree structure diagram, to be represented by a collection of followmaplets, known as a *followmap*. The next section describes how followsets, followmaplets and followmaps are generated from process structure diagrams. Before that, a brief overview is given of the constraints that need to be enforced in the architecture of process structures in order to support this transformation.

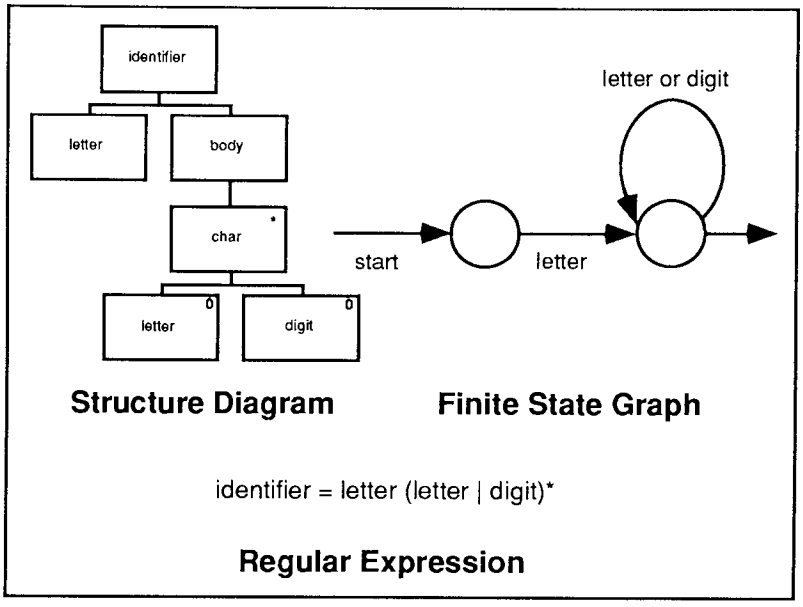


Figure 5.2. Different representations for the definition of an identifier.

Constraints on Process Structures

In order to realise a process as a followmap, several constraints need to be imposed on the way processes are specified; these are:

i. Primitive operations, which describe the low-level behaviour of a process, cannot appear in a process structure in arbitrary positions. Operations can be associated only with leaf nodes.

ii. If a read operation is to be used in a leaf node, then it must be the last operation of that leaf node (except for the very first read of the process, which is removed by inversion). This implies that only one read (or none) can appear in a leaf node.

iii. Null leaf nodes, which occur as alternatives in certain kinds of two-way selection, cannot have any primitive operations associated with them.

A few comments regarding these constraints are in order. Only allowing operations to be assigned to leaf nodes in the tree structure results in the process's potential general states being represented by leaf nodes, making the application of followset-based transformations much simpler. Also, it could be argued that it makes process specifications more understandable and maintainable by separating the concerns of specification into two distinct levels. The first level being its overall structure and the second level describing the sequential ordering of primitive operations associated with particular leaf nodes. Note that no kind of selective or iterative constructs can be incorporated within leaf node operations, all of which are to be regarded as primitive. It should be noted that these constraints impose no limitations on what processes can be specified i.e. all processes are still specifiable within these constraints. In fact, it is a practice that is recommended to improve the maintainability of specifications. Others such as [Poo91] and [Borgers90] have proposed different types of constraints, which again promote better maintainable specifications.

The realisation of the standard 'read-ahead' technique is reflected in the second constraint. Read-ahead is a strategy used for allocating reads to processes and reflects the requirement that a process must wait for, and receive each message from, an associated datastream before it can determine which next piece of text is to be executed [Jackson83].

Note that the general states in which a process can exist are defined to correspond to those points in a process's representation which contain 'read datastream', 'write datastream' and 'get state vector' operations [Jackson83]. This restricts the possible values an inspecting process can obtain when performing a state vector inspection but guarantees those values to be coherent. However, the usual restriction on state representation has been slightly relaxed here. A process's general states are considered to correspond to all its leaf nodes, whether or not those nodes have any associated reads, writes or get state vector operations.

Finally, being unable to associate operations with null leaf node components (depicted by a dash '—' in a selection leaf node) is an unavoidable consequence of the followset rules derived (see next section). However, when the actual usage of null nodes is considered, this restriction is not significant, at least for model processes. Null leaf nodes when used in model process structures represent 'events not happening'. Since a model process's execution is (mainly) dependent upon events in the real world triggering its activity, events which do not take place cannot, obviously, be captured and advance the process's execution.

5.2 *Followset Generation*

FIRST and FOLLOW

Generating followsets from process specifications is based on the successive application of two functions `FIRST` and `FOLLOW` (used in the context filter process depicted in Figure 5.1). `FIRST` and `FOLLOW` are repeatedly applied to a process structure as it is walked in pre-order fashion. During the walk, sets of symbols representing leaf node general states in the process structure are generated. The definitions

of FIRST and FOLLOW vary according to node type. As can be seen by their specification in Figure 5.3 (based on that given by [Jackson83]), the two functions are mutually recursive; note that rules (a) and (f) represent the non-recursive cases and also the points where actual symbols are generated.

```
a. If the node is a leaf node then
        FIRST(Parent) = {Parent}
b. Sequence components
   i    FIRST(Parent) = FIRST(Child[1])
   ii.  FOLLOW(Child[last]) = FOLLOW(Parent)
   iii. FOLLOW(Child[j]) = FIRST(Child[j + 1]), for j = 1 .. last - 1
c. Selection components
   i.   FIRST(Parent) = FIRST(Child[1]) or FIRST(Child[2]) or ... FIRST(Child[last])
   ii.  FOLLOW(Child[j]) = FOLLOW(Parent), for j = 1 .. last
d. Iteration components
   i.   FIRST(Parent) = FIRST(Child) or FOLLOW(Parent)
   ii.  FOLLOW(Child) = FIRST(Child) or FOLLOW(Parent)
e. If the node is a null leaf node then
        FIRST(Parent) = FOLLOW(Parent)
f. If the node is the root node then
        FOLLOW(Parent) = {}
```

Figure 5.3. Rule definition of FIRST and FOLLOW.

Figure 5.4 gives an example of applying different FIRST and FOLLOW rules to different node types. Note that node types are viewed differently depending upon whether FIRST or FOLLOW is being used. For example, node3 is to be viewed as an iteration when used as the argument of FIRST, but, when used as the argument of FOLLOW, node3 is viewed as a sequence component; in the first case, rule (d)(i) applies whereas in the second case rule (b)(ii) applies.

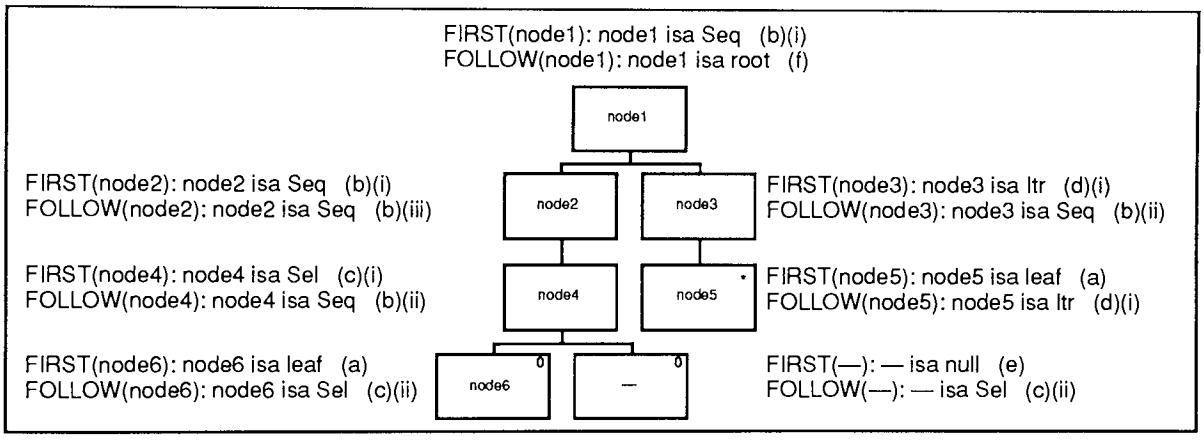


Figure 5.4. FIRST and FOLLOW for different node types.

It is possible to derive followsets of all nodes in a tree using these rules. However, not all of these sets are needed because of the constraint that primitive operations are only associated with leaf nodes. The only sets required are the first set of the root node and the followsets of the leaf nodes. The algorithm for followmap generation of a process ρ can thus be described as:

```
followmap(ρ)    =    followmaplet(root)
                +    followmaplet(leaf[1])
                +    followmaplet(leaf[2])
                +    . . . . . .
                +    followmaplet(leaf[n])
```

As an example of applying this simple algorithm, the followmap `followmap(node1)`[1] of the tree structure from Figure 5.4 has been 'traced' below:

```
FIRST(node1)      =    FIRST(node2)                      - (b)(i)
                  =    FIRST(node4)                      - (b)(i)
                  =    FIRST(node6) or FIRST(null)       - (c)(i)
FIRST(node6)      =    {node6}                           - (a)
FIRST(null)       =    FOLLOW(null)                      - (e)
                  =    FOLLOW(node4)                     - (c)(ii)
                  =    FOLLOW(node2)                     - (b)(ii)
                  =    FIRST(node3)                      - (b)(iii)
                  =    FIRST(node5) or FOLLOW(node3)     - (d)(i)
FIRST(node5)      =    {node5}                           - (a)
FOLLOW(node3)     =    FOLLOW(node1)                     - (b)(ii)
                  =    {}                                - (f)

FOLLOW(node6)     =    FOLLOW(node4)                     - (c)(ii)
                  =    FOLLOW(node2)                     - (b)(ii)
                  =    FIRST(node3)                      - (b)(iii)
                  =    FIRST(node5) or FOLLOW(node3)     - (d)(i)
FIRST(node5)      =    {node5}                           - (a)
FOLLOW(node3)     =    FOLLOW(node1)                     - (b)(ii)
                  =    {}                                - (f)

FOLLOW(null)      =    FOLLOW(node4)                     - (c)(ii)
                  =    FOLLOW(node2)                     - (b)(ii)
                  =    FIRST(node3)                      - (b)(iii)
                  =    FIRST(node5) or FOLLOW(node3)     - (d)(i)
FIRST(node5)      =    {node5}                           - (a)
FOLLOW(node3)     =    FOLLOW(node1)                     - (b)(ii)
                  =    {}                                - (f)

FOLLOW(node5)     =    FIRST(node5) or FOLLOW(node3)     - (d)(ii)
FIRST(node5)      =    {node5}                           - (a)
FOLLOW(node3)     =    FOLLOW(node1)                     - (b)(ii)
                  =    {}                                - (f)
```

From the trace, it is possible to deduce the next set of valid general states to follow any given general state in the process structure by evaluating `FOLLOW(givenState)`, where `givenState` is represented by a leaf node. This ability to derive systematically the next valid general state set can be used as the basis for realising the operational semantics of processes. From a trace such as the one above, all the followmaplets can be derived; these are:

```
FIRST(node1)     = {node6, node5}
FOLLOW(node5)    = {node5}
FOLLOW(node6)    = {node5}
```

The first followmaplet is for the root node and represents the first general state in which the specified process above can be in; the other two followmaplets are for its leaf nodes, representing all other possible general states. Note that the followmaplet `FOLLOW(-)` does not appear above. This is because null nodes do not get entered into followsets (see rule (e) above) and hence there is no need for `FOLLOW(-)` to be in a process's followmap. A more comprehensive example of tracing the `Book` process structure (see Figure 3.2 in [3.2]) is given in Appendix C.

[1] Note that the process name is usually the name of the root node.

5.3 Guards

Run-time Behaviour

Followsets are insufficient in themselves to realise the complete behaviour of a process, since they merely give a static representation of the process. It is also necessary to be able to determine which actual state in a set of states will follow a given state during a process's execution. This run-time semantics can be derived by incorporating extra processing in the followmap generation algorithm.

Operationally, in order to traverse from one leaf node state (source) to a following leaf node state (destination) in a process structure, it must be true that on reaching this destination all conditional logic (iterations and selections) passed through en route have been evaluated. For example, in the simple process structure diagram of Figure 5.5, the followset associated with leaf node B is {D, F, G}. In order to traverse from B to G, denoted as p(B, G), the associated conditional logic at D and F (the conditions C1 and C2 respectively) must be false. This conditional logic for traversing from one source state to another (valid) destination state, known as a *path* [Roper87], is realised as a *guard.*

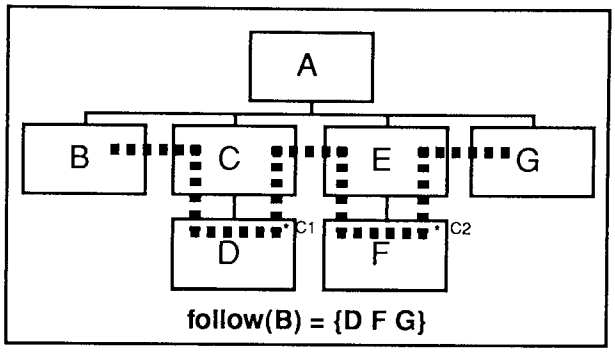


Figure 5.5. The path p (B, G) through process A.

Attached to each general state in every followset of general states is a guard representing the conditional operations associated with selections and iterations in a process's structure which would be evaluated in traversing the path from some given source state to that state. In the above example, using the followmaplet FOLLOW(B) = {D, F, G}, the members of this followset have three guards g(B, D), g(B, F) and g(B, G) respectively. Each guard, when evaluated, indicates whether or not the associated destination state is the next valid state to follow the source state. Because JSD process semantics is completely deterministic[2], only one guard associated with any particular followset can be true at any one time (although see section [5.4] the subsection entitled 'null nodes'). By a simple modification to the two functions FIRST and FOLLOW, guards can be derived automatically *in situ* of followset generation.

Guard Generation

As a tree is walked during the application of FIRST and FOLLOW, the path taken will inevitably traverse through nodes which are iterations and selections. Each of these node types will reference its attendant boolean logic. When encountered, the conditions at these nodes are conjoined to form composite conditionals. These composite conditionals represent the guard functions for traversing the path from source states to destination states. It should be noted that, for iteration nodes, the negation of the iteration condition is conjoined because, obviously, to exit from an iteration its condition must be false. Another

[2] This is for a structurally valid process where all conditions in each selection are mutually exclusive.

way of viewing this situation is to note that what can follow the last leaf node within an iterative subtree, is either the first leaf node of the iterative subtree or (when the controlling condition for the iteration becomes false), some other node outside the iterative subtree. For example, in the process structure shown in Figure 5.6, there exists an iterative subtree at `node3`, where the leaves of that subtree are `node5`, `node6` and `node7`. Using the followmaplet `FOLLOW(node7) = {node5, node8}`, the two associated guards would be `g(node7, node5)` and `g(node7, node8)`. This second guard would be:

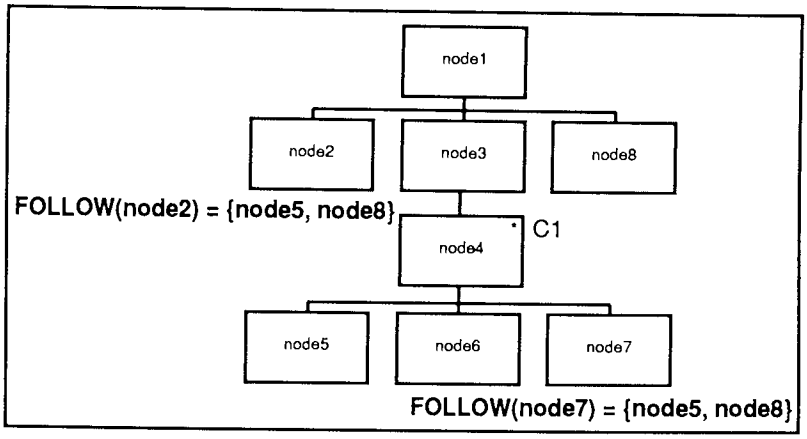


Figure 5.6. Guard generation for iteration nodes.

```
g(node7, node8) = not C1
```

Similarly, consider `FOLLOW(node2)`. In order to get from `node2` to `node8` the condition `C1` must be false. The guard for this state change would be:

```
g(node2, node8) = not C1
```

5.4 *General implementation of Followsets and Guards*

Direct Realisation

A convenient way to show how a process can be realised in terms of followsets and guards is to give an example of the result of the followmap generation algorithm being applied to a process. Using the `Book` process structure (see [3.2]), its followmap based representation (Figure 5.7) is presented in a pseudo Pascal type syntax. A name with the 'c' subscript, e.g. $\texttt{loan}_c$ represents the condition at that named node in the process structure, i.e. in this case, the condition at `loan`. The complete condition before each **`then`** keyword is an actual guard. The operation *`process`* `<name>` signifies invocation of the primitive operations associated with that leaf node.

```
process bookProcess;
   var currentState: bookStates;
begin
   currentState := book;
   while true do
   begin
      {book section}
      if (currentState = book) and true
                        then currentState := acquire; process acquire;
      {acquire section}
      elseif (currentState = acquire) and true
                        then currentState := classify; process classify;
```

```
        {classify section}
        elseif (currentState = classify) and loan_c
                              then currentState := lend; process lend;
        elseif (currentState = classify) and not loan_c and sell_c
                              then currentState := sell; process sell;
        elseif (currentState = classify) and not loan_c and dispose_c
                              then currentState := dispose; process dispose;
        {lend section}
        elseif (currentState = lend) and renew_c
                              then currentState := renew; process renew;
        elseif (currentState = lend) and not renew_c
                              then currentState := return; process return;
        {renew section}
        elseif (currentState = renew) and renew_c
                              then currentState := renew; process renew;
        elseif (currentState = renew) and not renew_c
                              then currentState := renew; process return;
        {return section}
        elseif (currentState = return) and loan_c
                              then currentState := lend; process lend;
        elseif (currentState = return) and not loan_c and sell_c
                              then currentState := sell; process sell;
        elseif (currentState = return) and not loan_c and dispose_c
                              then currentState := dispose; process dispose;
        {sell section}
        elseif (currentState = sell)
                              then null;
        {dispose section}
        elseif (currentState = dispose)
                              then null
    endWhile
endProcess;
```

Figure 5.7. Format of the transformed `Book` *process,* `followmap(bookProcess)`.

One point in need of explanation is why some of the guards have `and true`. This is because the value for the guard associated with traversing the path from one sequence node to its immediate brother sequence node is simply true, as there exists no conditional logic to traverse. Guard components which are simply `and true` can of course be dropped. Note that `bookProcess` will continuously loop, eventually 'resting' forever in a `sell` or `dispose` state.

The realisation of the above for any process is achieved simply by enumerating through all the followmaplets of the process's followmap. For each state mapping of the form:

$$\texttt{FOLLOW(start)} = \{\texttt{fs}_1, \texttt{fs}_2, \ldots, \texttt{fs}_n\}$$

the following is produced:

```
        elseif currentState = start and g(start, fs_1)
                              then currentState := fs_1; process fs_1;
        elseif currentState = start and g(start, fs_2)
                              then currentState := fs_2; process fs_2;
        elseif currentState = start and g(start, fs_n)
                              then currentState := fs_n; process fs_n;
```

where `g(start, fs`$_n$`)` is expanded into a boolean expression or is made a function which returns a boolean value. For the very first component of the multi-way `if`, the followmaplet of `FIRST(root)` is used since this represents the first possible state in which a process can be.

State Dismemberment

An important point to be highlighted about the followmap representation of a process is that it is effectively the original process dismembered into all its leaf node states. This form of dismemberment is known as *process state dismemberment.* A distinguishing feature of this form of dismemberment is that each dismembered component has no structure (iterations and selections) within it with respect to the original process.

As the process has been process state-dismembered, the original process's internal structure (i.e. non-leaf nodes) has been absorbed within the guards and the followsets themselves. Since internal structure has now disappeared, the problem of jumping into a block structure (e.g. a selection or a while loop) which exists with code generated from transformations discussed in [3.4] and [4.3] no longer exists. Iterations and selections have effectively disappeared via the dismemberment and become a set of node states with associated guard functions. In effect, the entire original process structure can be viewed as one large flat multiple selection. No matter how complicated the original process, after destructuring via the followmap algorithm, the result is reduction of the process to the simplest structure possible, an iteration of multiple selections — see Figure 5.8.

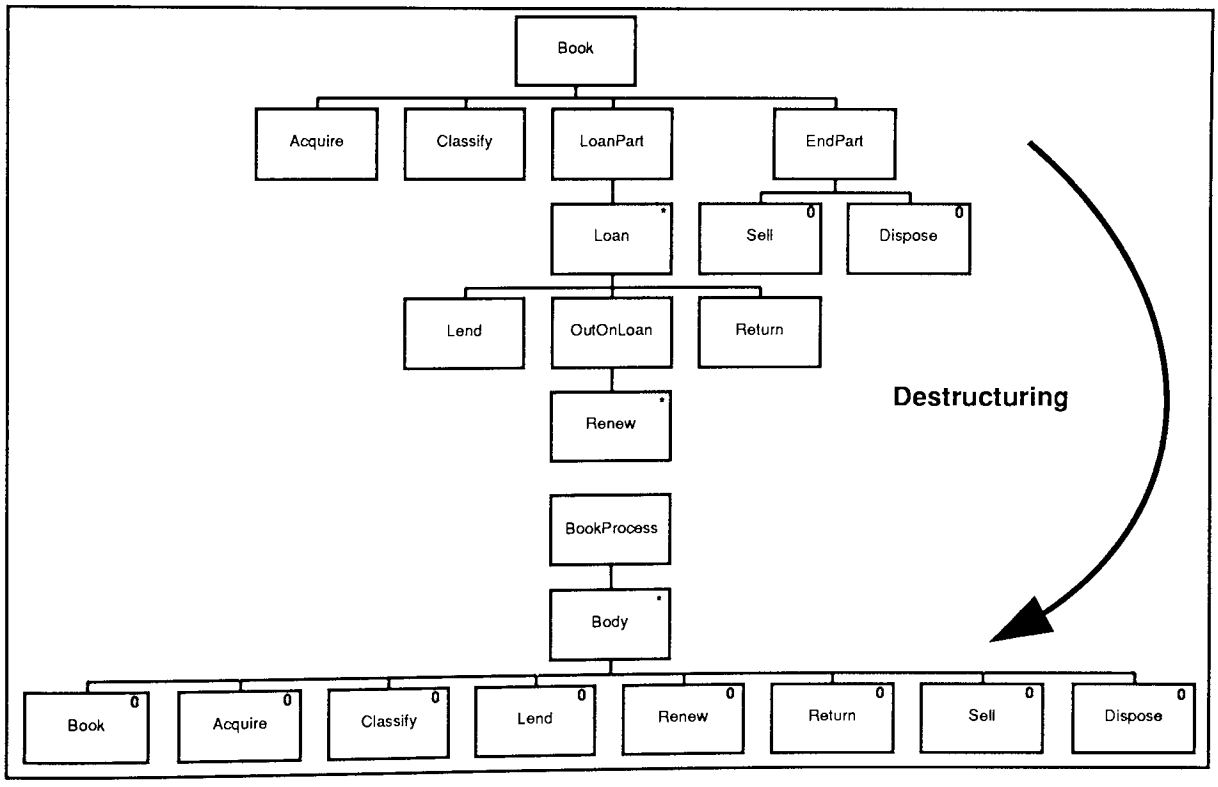


Figure 5.8. Destructuring of process structures.

It should be noted that the resulting structure is not a true structure as defined by [Jackson83], and is used here only for illustrative purposes; for example, it is clearly not permissible for `Acquire` to be executed more than once, which the destructured process shown in Figure 5.8 in principle permits.

Null Nodes

So far, the order in which the dismembered components within the realised algorithm have appeared has been the same as the associated general states appear in the process structure when traversed in a pre-order fashion. This ordering has not been accidental and in fact is critical. The reason for this is that when a process structure has a null node within it, the potential exists for two guards to be true at the same time. This can be demonstrated using the following two (near identical) process structures (Figure 5.9).

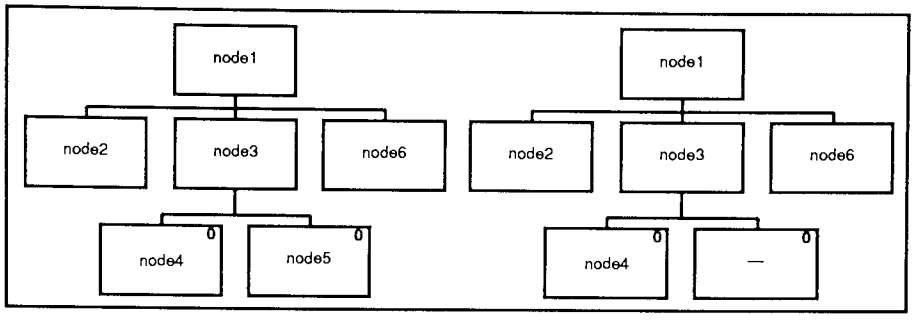


Figure 5.9. Null nodes and guard generation.

It should be noted that in the first structure of Figure 5.9, `FOLLOW(node2) = {node4, node5}`; guard `g(node2, node4)` would be equal to the condition associated with `node4` and `g(node2, node5)` would be equal to the condition associated with `node5`. However, in the second structure, `FOLLOW(node2) = {node4, node6}`. Null nodes do not appear in followset entries as can be seen from the followset rules (rule (e) in [5.2]). When the guards for `FOLLOW(node2)` in the second structure are considered, a problem arises. `g(node2, node4)` is the same as for the previous structure i.e. equal to the condition associated with `node4`, but `g(node2, node6)` will always return true. The reason for this is that `node6` is regarded as a sequence node during guard generation, and the default guard condition for all sequences is simply true. Clearly, if guards associated with a particular state are evaluated in any arbitrary order, the potential exists for a guard to initiate the execution of the wrong dismembered component, e.g. code associated with `node4` in the above process structure might never be executed simply because the guard associated with `node6` was evaluated first. If the order of guard evaluation is the same as that in which leaf node general states appear in the process structure, this potential problem never arises.

Optimisations

Representing a transformed process as a collection of dismembered components has the disadvantage of being an inefficient implementation. This can be demonstrated in the following way. Suppose a process structure is large and finishes with an iteration. The problem which arises is that on each cycle of the (now destructured) iteration within the generated code, all the previous dismembered parts will have their guards evaluated, until processing again reaches the last component — see Figure 5.10.

Use of a `case` statement, (unfortunately not available in Smalltalk-80) can be seen as a form of optimisation. It saves having to repeat within the **`elseif-then`** branches every test for the `currentState` and, more importantly, reduces the search space for detecting the next dismembered part to be evaluated — in an implementation using a `case` construct, all guard start states which are the same are collected together. For example, the followmap representation of the `Book` process could now be (Figure 5.10):

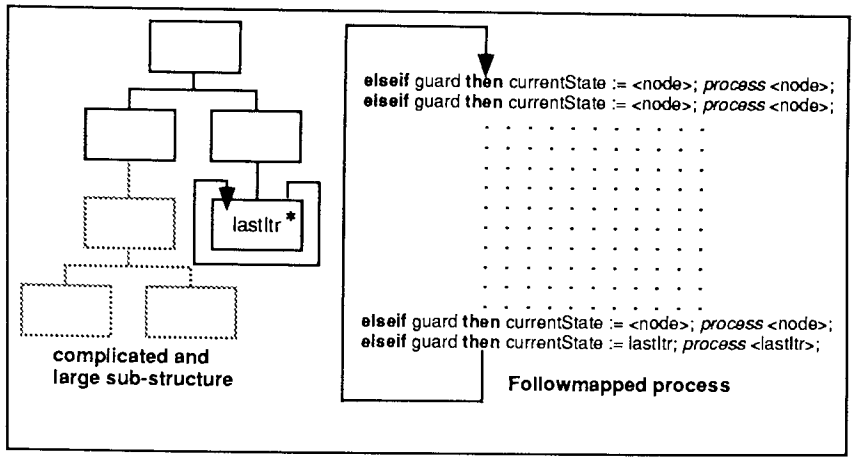


Figure 5.10. Worst case scenario for a followmapped process.

```
process bookProcess;
  var currentState: bookStates;
begin
  currentState := book;
  while true do
  begin
    case currentState of
      book:     currentState := acquire; process acquire;
      acquire:  currentState := classify; process classify;
      classify:begin
                 if loanc then
                    currentState := lend; process lend;
                 elseif not loanc and sellc then
                    currentState := sell; process sell;
                 elseif not loanc and disposec then
                    currentState := dispose; process dispose
               end;
      lend:    begin
                 if renewc then
                    currentState := renew; process renew;
                 elseif not renewc then
                    currentState := return; process return
               end;
      renew:   begin
                 if renewc then
                    currentState := renew; process renew;
                 elseif not renewc then
                    currentState := return; process return
               end;
```

```
      return:   begin
                  if loanc then
                     currentState := lend; process lend;
                  elseif not loanc and sellc then
                     currentState := sell; process sell;
                  elseif not loanc and disposec then
                     currentState := dispose; process dispose
                end;
      sell:     null;
      dispose:  null
    endCase
  endWhile
endProcess;
```

Figure 5.11. Followmap optimisation using a `case` structure.

Note also that the superfluous `and true` guards have been removed.

5.5 *Asynchronicity and Datastreams*

Inversion Revisited

The transformation scheme discussed so far has yet to address the issue of datastreams. However, it should first be noted that read operations in a process structure can be assigned only at the leaf node level — the allocation of read points takes place during the specification phase of JSD, when the standard read-ahead technique is used. As has been discussed at the start of this chapter, one of the constraints imposed on a followset-based transformational approach is that leaf nodes have only a single (or no) read associated with them. When a read does occur, it must be the last operation occurring in a particular leaf node. A followmapped process thus results in all the read points of that original process being at the end of each dismembered component — no dismembered component contains within it any sub-structure, as already shown, and so will not contain other reads. The issue at hand is how to deal with these reads to obtain inversion.

It has previously been shown (see [3.4]) that the inversion transformation produces an architecture of reentrant subroutines which are implemented in procedural languages by sending control to different parts of the procedure via a multi-way `goto` instruction (see [4.3]). In Smalltalk-80, inversion can be realised using context manipulation although, as already discussed, this approach is specific to that language. `goto`'s are required in procedural languages to support the suspend and resume mechanism needed to simulate coroutine behaviour. Read operations are deleted and become entry points into subroutines. Data originally obtained by read operations is now passed to an inverted subroutine via its parameter list. When the subroutine is called, the multi-way `goto` guarantees that code associated with the next valid leaf node state is executed by resuming execution from the last suspension point. It is clear that such a scheme executing specific portions of code is almost identical, from an operational point of view, to that of a followmapped process, except that the 'scanning' activity to find the next valid piece of code to execute is carried out by the multi-way `goto`.

```
procedure book(var dsRecord: bookRecord);
  var currentState: bookStates;
begin
  read := true;
  while read do
  begin
    case currentState of
      book:     currentState := acquire; process acquire; read := false;
      acquire:  currentState := classify; process classify; read := false;
      classify:begin
                 if loanc then
                   currentState := lend; process lend; read := false;
                 elseif not loanc and sellc then
                   currentState := sell; process sell;
                 elseif not loanc and disposec then
                   currentState := dispose; process dispose
               end;
      lend:    begin
                 if renewc then
                   currentState := renew; process renew; read := false;
                 elseif not renewc then
                   currentState := return; process return; read := false
               end;
    . . . . . .
    endCase
  endWhile
endProcedure;
```

Figure 5.12. An inverted followmapped process.

In order to suspend this continuous looping around the **case** construct, each read at the end of a relevant dismembered component needs to be replaced by a termination of the iteration. This can be done by simply setting a boolean flag, and making a test for its value at the start of the iteration. Data originally retrieved by the read operation is now passed, as before, via a parameter to the transformed subroutine. This result is inversion effected on the followmapped process (see Figure 5.12 above).

After state vector separation is applied to the followmapped and inverted process, an entire state (i.e. state vector) can be passed to the newly generated subroutine, which then assumes a particular process identity, executes some guarded code, and finally suspends. Although on each cycle of the main loop, many unnecessary guards may be evaluated in order to reached the desired state, the transformation guarantees that the next valid dismembered component will be evaluated. An alternative view of the structure and behaviour of a transformed followmapped process (i.e. inverted and state vector separated) is that of an *event manager* akin to that found in the main event loop of the MacApp framework used for building applications on Apple Macintosh computers [Apple90]. The operation of this main event loop (found in class `TApplication`) is to determine what behaviour the machine is to carry out depending upon the current event (implemented by the method `DispatchEvent`). The behaviour of a transformed process is to determine which dismembered component is to be evaluated depending upon the current state of the process — both are similar in behaviour.

Since it is possible via the followmap approach to realise inversion without having to use a `goto` primitive, JSD specifications can therefore be realised in languages which do not support the `goto` primitive and do not allow the manipulation of the host machines run-time stack — in other words, the transformational approach has been made more general. Clearly, the use of followsets to transform JSD specifications is sufficiently general for it to be applied to all general purpose procedural languages. The next chapter shows how this transformational approach is realised in a pure object oriented language like Smalltalk-80.

A selection of good tools is a fundamental requirement for anyone contemplating the maintenance and repair of a motor vehicle. For the owner who does not possess any, their purchase will prove a considerable expense, offsetting some of the savings made by doing-it-yourself. However, provided that the tools purchased are of good quality, they will last for many years and prove an extremely worthwhile investment.

Haynes Owners Workshop Manual — 1984

6 Smalltalk-80 Implementation and Tool Support

6.1 Further Transformations

State Vector Separation

The previous chapter concentrated on a realisation of the inversion mechanism by a followset-based transformation. This chapter describes how that transformation is realised in Smalltalk-80. Note, however, that although an alternative approach for realising inversion has been presented in Chapter 4, the approach was abandoned due to its dependence on system stack-frame manipulation, which Smalltalk-80 permits but most other object oriented languages do not. Before presenting the implementation details of the transformations and the tool which has been built to support them (also in Smalltalk-80), an overview of the other mappings needed to realise JSD specifications in an object oriented language will be discussed.

It should be recalled that a process class in a network specification represents all possible instances of that class. Each process instance has its own unique identity and is generally in a different state to its neighbours. Potentially, many thousands of these identical processes can exist in a network. To reduce this multiplicity, the transformation of state vector separation is applied to each individual process resulting in one copy of the process text and many copies of the state vectors — the individual states of each process.

Basically, each process is transformed by removing the (persistent) variable declarations from the process's specification. This collection of process variables is used as the basis for describing a database's record structure. In addition to the declared variables, a variable (usually called QS) that is used to store the resumption points of the inverted process is also included. Each such state vector becomes an entry in the state vector database (see SVDB in Figure 6.1 overleaf); thus, for every process instance there will be an entry in the SVDB. This arrangement enables each process instance to run on the same processor by using the reentrant subroutine, produced by inversion, to update a loaded state vector during its execution. When the subroutine is called, its first activity is to load the state vector of the process instance whose identity it is to assume on that call; as an alternative, the subroutine could be passed the

specific state vector required as a parameter. At the subroutine's next suspend point, the (possibly updated) state vector is written back to the state vector database.

One obvious influence in deciding how to implement state vector separation in an object oriented language like Smalltalk-80 can be found within the implementation of such a language itself. When an instance of an object is created, the only parts of its class which are duplicated are its instance variables. Each object points to its class when needing to access behaviour. This reflects to a large degree the objective of state vector separation. To take advantage of this feature of the language, JSD process state variables are realised as instance variables of Smalltalk-80 classes. For example, the process variables of the `Book` process (see [3.2]) include:

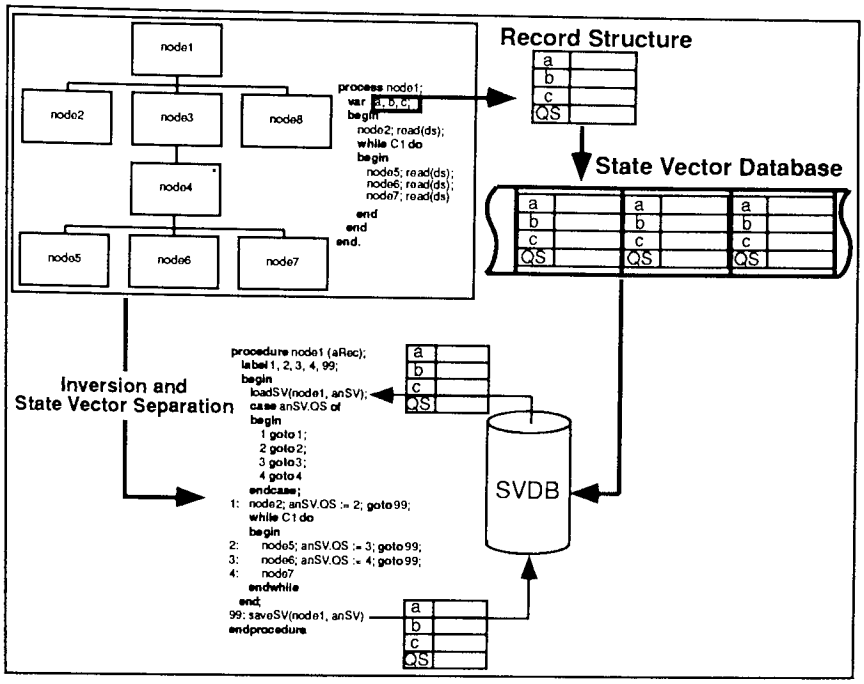


Figure 6.1. The separation and storage of state vectors.

```
dateAcquired, title, purchasePrice, lastBorrower, lendCount, lendDate,
dateOfLastRenewal, inLibrary, totalTimeOnLoan
```

These would become the instance variables of the class `BookProcess` shown below:

```
JSD subclass: #BookProcess
    instanceVariableNames:    'dateAcquired title purchasePrice lastBorrower
                               lendCount lendDate dateOfLastRenewal inLibrary
                               totalTimeOnLoan '
    classVariableNames:       ''
    poolDictionaries:         ''
    category:                 'JSD-Process'
```

An alternative scheme would be to realise JSD processes as methods, and the process's state variables as parameters and local variables of these methods. However, this approach would be untenable. A way of reducing the number of identical copies of these methods would be required, i.e. a mechanism for state vector separation would be needed. This would involve the implementer developing what was already in effect present within the run-time environment and hence 're-inventing the wheel'. Directly exploiting the class/instance mechanism, which is common to most object oriented systems, is the obvious approach to adopt.

To effect state vector separation, all JSD process classes within a network specification are transformed to individual Smalltalk-80 classes which are themselves all subclasses of an abstract class called `JSD`. The name of each class is derived from a process's name, i.e. the root node in the process's structure diagram. For example, the `Book` process is realised as the class `BookProcess` (see the example subclass definition above). The extension of `Process` guarantees that the class name does not conflict with any other in the Smalltalk-80 environment.

Since process classes in JSD networks are realised as actual classes in Smalltalk-80, all operations in a process class are realised as instance methods. Conditional operations for iterations and selections in a process have a one-to-one mapping with methods. These methods always return a boolean value. Their names are derived from the node in the process structure representing the iteration or selection, but with the extension of `C`. For example in the `Book` process, the four conditional nodes' operations are represented as the four methods: `loanC`, `renewC`, `sellC`, `disposeC`.

Method granularity for leaf operations is such that a single method encapsulates all of a given leaf's assigned operations. Names of such methods are the same as the names of the leaf nodes themselves. However, as it is not possible to have within the same class two methods with the same name, it is therefore not possible to represent a transformed process structure with two leaf nodes with the same name. One simple way to resolve this problem would be to assign a unique index to each leaf node during a pre-order walk of the tree, thereby making it possible to distinguish between nodes of the same name. This would also overcome another problem associated with non-unique names for process leaves which relates to the generation of followsets. Although it is possible for two or more leaf nodes with the same name to follow some other node in a followset, only one instance of any node is entered by the followset generation algorithm since a followset is simply a set (albeit with an ordering defined over its members).

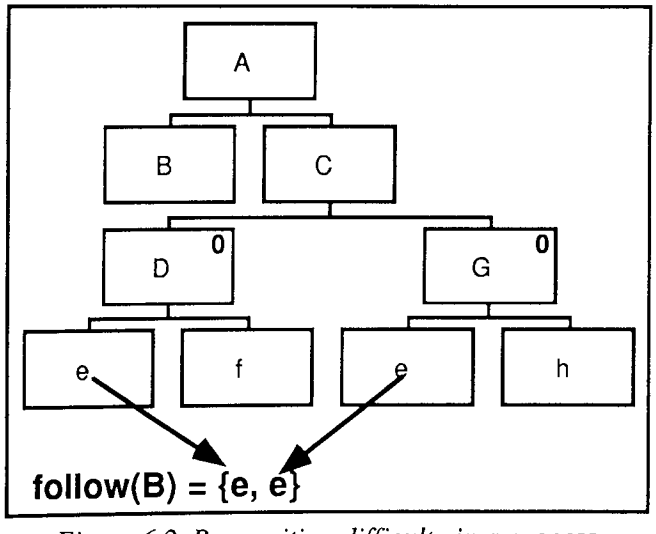


Figure 6.2. Recognition difficulty in a process.

As regards the problem of non-unique node names, it is interesting to note that a byproduct of the followset generation algorithm is its potential for identifying *recognition difficulties*. If a situation occurs whereby several nodes with the same name can follow another node, a recognition difficulty has arisen. Recognition difficulties are problems which arise usually in the modelling phase of JSD where an event in the real world cannot accurately be trapped and separated from other events. For example, consider the following simple process structure in Figure 6.2.

Here the event `e` occurs twice in the structure. The problem is that both `e`'s can follow node `B` and so it is not possible to determine which branch the process should next enter i.e. `D` or `G`. Recognition difficulties are solved by one of two techniques. The first is *multiple read ahead* [Jackson75] where the process executes two (or more) consecutive reads (in this case at node `B`). The second (buffered) read will encounter either an `f` or `h` message thus resolving the recognition difficulty. However, multiple read ahead is not always applicable, e.g. when the number of extra reads required becomes excessive or indeterminate. Also, multiple read-ahead is rarely used in JSD[1] and instead the more popular approach is to use *backtracking*. Briefly, a process structure incorporating a backtracking solution is one where the designer of the process structure *posits* that a certain sequence of events will occur — for example an `e` followed by `f`; if at node `f` an `h` is read instead, then that branch of the structure is *quit* and the *admit* part (the `G` branch) of the process structure entered (see [Jackson75, Jackson83]).

Format of Process Classes

As all processes need to store their unique identification and their resumption point, two variables common to all process classes have been defined in the abstract class `JSD` called `key` and `QS` respectively. The values which `QS` holds are instances of the class `Symbol`; the names of these symbols are derived from leaf nodes in a process structure , (recall that leaf nodes represent all possible general states of a process). Another important instance variable of class `JSD` called `suspended` signifies whether an inverted process has reached a suspend point.

Two class instance variables called `instances` and `followMap` are defined in the abstract class `JSD`. These two variables enable all subclasses of `JSD` to have slots for storing all their instances and to share their followset-based representation. The variable `instances` always references an instance of class `SVDB` (a subclass of `Dictionary`, which is part of the Smalltalk-80 environment), into which all instances of the class can be deposited. The variable `followMap` references an instance of class `FollowMap` which is essentially a dictionary of `FollowMaplet` instances. A `FollowMaplet` is a

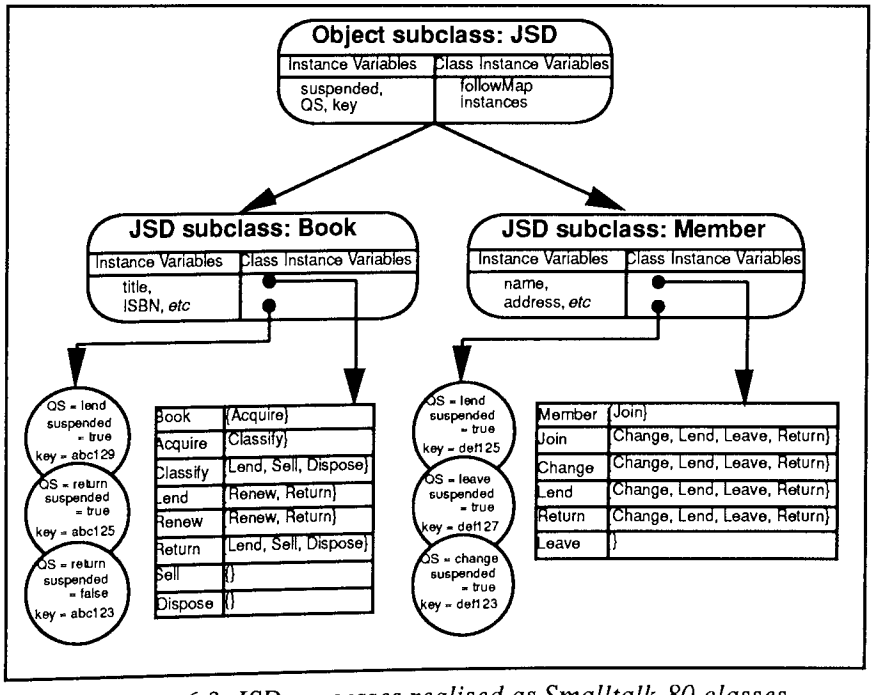


Figure 6.3. JSD processes realised as Smalltalk-80 classes.

[1] My thanks to John Cook of LBMS for bringing this point to my attention.

mapping of a start state to a set of follow states, this set being an instance of class `Followset`. Hence, the variable `followMap` holds all the followsets belonging to the owning class. The previous diagram (Figure 6.3) gives a schematic overview of the class instance and instance variables used. In summary, the overall mappings (highlighted in Figure 6.4) are:

- Process classes in the network are mapped to classes in the Smalltalk-80 environment, which are themselves subclasses of the abstract class `JSD`.
- Process state variables are mapped to instance variables of a class.
- Leaves of process structure diagrams are mapped to methods.
- Conditions of iterations and selections are mapped to methods.
- A process's followset based representation and a simple realisation of a state vector database are stored in class instance variables.

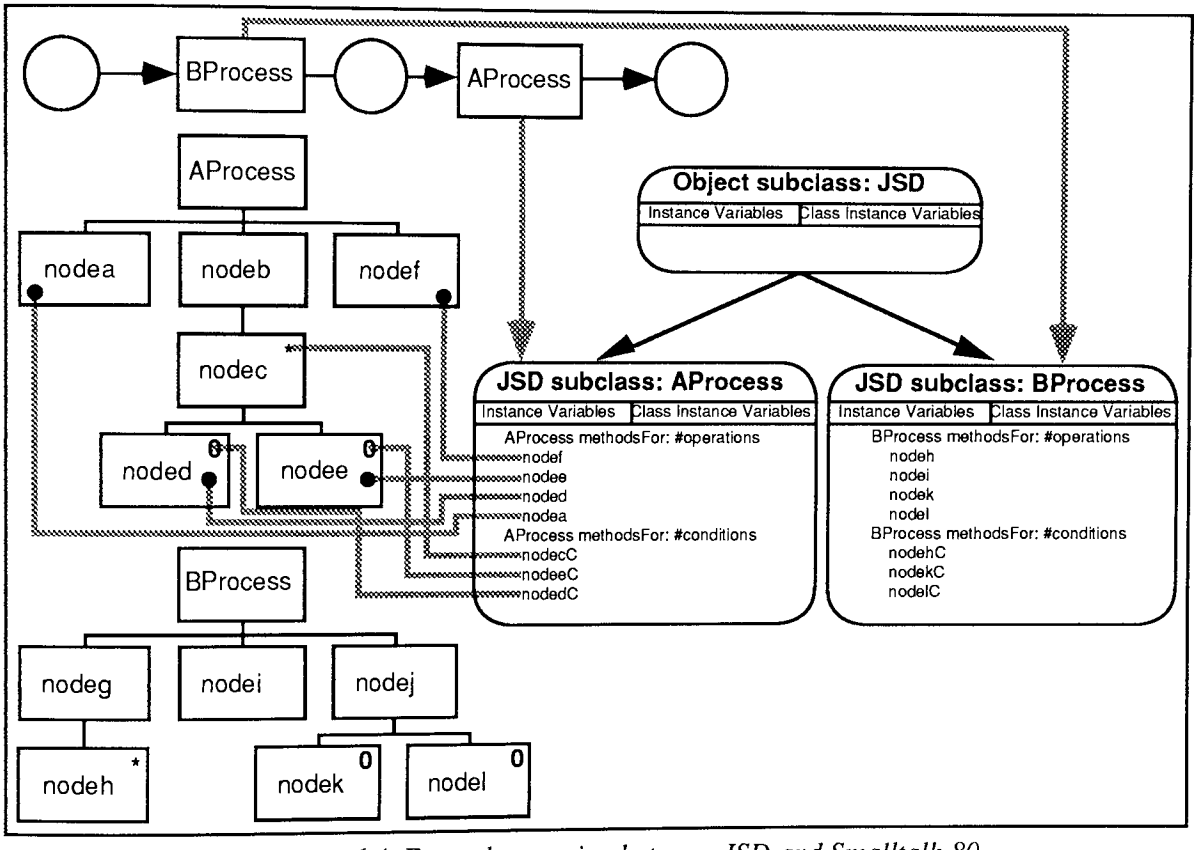


Figure 6.4. Example mapping between JSD and Smalltalk-80.

A complete listing of the Smalltalk-80 representation of the `BookProcess` can be found in Appendix D.

6.2 *Generation of Followmaps*

Process Structures

In order to present an overview of how the two functions `FIRST` and `FOLLOW` are realised in Smalltalk-80, it is first necessary to describe how process structure diagrams are represented. Four distinct classes are used to represent nodes in a process tree, namely `Node`, `ItrNode`, `SelNode` and `SeqNode`. `Node` is an

abstract class and the other three are subclasses of it. Each node type in the tree is represented by the appropriate instance of one of these three concrete subclasses. A tree is simply a linked structure composed of instances of these concrete node classes. Each node instance has a pointer field for its potential parent, left-sibling, right-sibling, and first-child, enabling nodes to be linked as shown in Figure 6.5. This approach to representing trees is based on [Wilson84]. Using this approach to node linkage, any JSD process structure diagram can be represented and traversed. Also having explicit pointers for parent nodes and first sons facilitates the generation of followsets and guards.

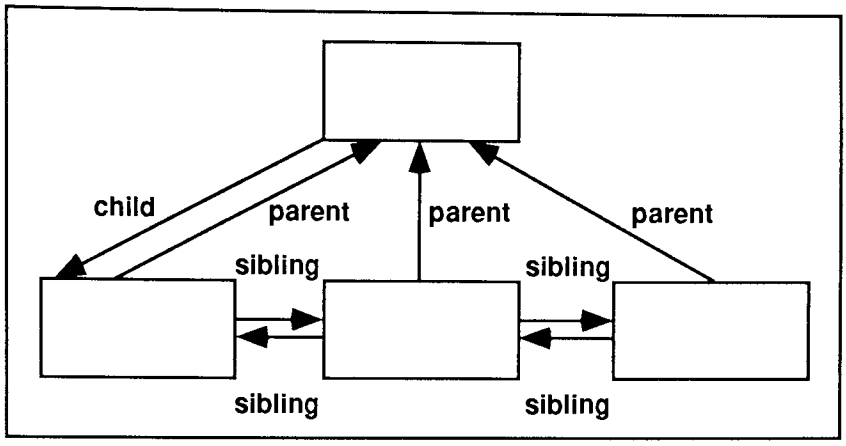


Figure 6.5. Node linkage in the representation of structure diagrams.

Within the four node classes described, four distinct message patterns are used in the generation of followsets:

`first:`, `follow:`, `nonLeafFirst:`, `followFromChild:with:`

The implementation of the messages `first:` and `follow:` are to be found in the abstract class `Node`. The other two messages are each in the three concrete classes `ItrNode`, `SelNode` and `SeqNode`, but have different implementations for each class. The parameter to all the messages is an instance of a `FollowMap` (the additional parameter for `followFromChild:with:` will be explained later). The methods `first:` and `follow:` in class `Node` factor out all that is common to generating followsets, e.g. testing for the occurrence of a root node, null leaf node, etc. However, although this reduces the amount of code, the names of the methods in the subclasses of `Node` representing the remainder of the `FIRST` and `FOLLOW` functionality have to be different to those defined in class `Node` itself to avoid unwanted method invocation.

Evaluation of `first:`

When an instance of one of the three concrete node classes is sent the `first:` message, its implementation is found in class `Node` via inheritance. `first:` determines whether or not its receiver is a leaf node. If it is not, then `first:` sends the message `nonLeafFirst:` to the original receiver. The implementations of `nonLeafFirst:` for `ItrNode`, `SeqNode` and `SelNode` are shown overleaf respectively in Figure 6.6; they are a direct translation of the rules for followset generation for the `FIRST` function given in [5.2]. Note that the message `firstSon` returns the receiver node's first child node and the message `children` returns a collection of all the receiver's immediate children.

```
nonLeafFirst: aFollowMap
        "FIRST(Parent) = FIRST(Child) or FOLLOW(Parent)"
        self firstSon first: aFollowMap
        self follow: aFollowMap
        ↑aFollowMap
nonLeafFirst: aFollowMap
        "FIRST(Parent) = FIRST(Child[1])"
        self firstSon first: aFollowMap
        ↑aFollowMap
nonLeafFirst: aFollowMap
        "FIRST(Parent) = FIRST(Child[1]) or FIRST(Child[2]) or …
                FIRST(Child[last])"
        self children do: [:aChild | aChild first: aFollowMap].
        ↑aFollowMap
```

Figure 6.6. Implementations of `nonLeafFirst:` *for the three node types.*

As can be seen, none of these three implementations makes any additions to the followmap being passed around. This is because the only time when an entry is added is when a receiver node is a leaf node; the addition of leaf node symbols to the followmap takes place within the implementation of `first:` itself. Note that in the event of a null leaf node being encountered, `first:` sends the receiver the `follow:` message (as null nodes do not have entries in followsets).

Evaluation of `follow:`

When an instance of one of the three concrete node classes is sent the `follow:` message, the implementation is again found in class `Node`. This determines whether or not the receiver of the `follow:` message is the root node. If it is not, then within the implementation of `follow:` the message `followFromChild:with:` is sent not to the original receiver but to the receiver's father node. The two parameters of the message are the receiver node (the extra parameter mentioned earlier) and the followmap being generated. The following (Figure 6.7) are the three implementations of `followFromChild:with:` for the three classes `ItrNode`, `SelNode` and `SeqNode` respectively:

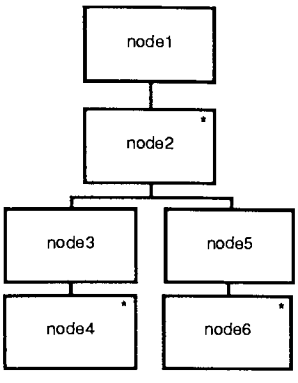
```
followFromChild: aNode with: aFollowMap
        "FOLLOW(Child) = FIRST(Child) or FOLLOW(Parent)"
        aNode first: aFollowMap.
        self follow: aFollowMap.
        ↑aFollowMap
followFromChild: aNode with: aFollowMap
        "FOLLOW(Child[j]) = FOLLOW(Parent) [j = 1 .. last]"
        self follow: aFollowMap.
        ↑aFollowMap
followFromChild: aNode with: aFollowMap
        "FOLLOW(Child[last]) = FOLLOW(Parent)
        FOLLOW(Child[j]) = FIRST(Child[j+1]) [j= 1 .. last-1]"
        aNode isLastRight
                ifTrue: [self follow: aFollowMap]
                ifFalse: [aNode rightBrother first: aFollowMap].
        ↑aFollowMap
```

Figure 6.7. Implementations of `followFromChild:with:` *for the three node types.*

As is evident, the implementations are a simple translation of the rules given in (see [5.2]).

Degenerate Case

Although the essential parts of the implementation for generating followsets have now been covered, the implementation so far would not work for some process structures. As has already been noted, the way followset generation is achieved is highly recursive. The exit conditions for this recursion are when a node is either a leaf node (in which case a symbol is entered into a followmap) or the root node (see rules in [5.2]). However, consider the trace of `FOLLOW(node4)` shown in Figure 6.8.




```
                                                          Rules in [5.2]
Follow(node4) =  First(node4) or Follow(node3)      (d) (ii)
First(node4)  =  {node4}                            (a)
Follow(node3) =  First(node5)                       (b) (iii
First(node5)  =  First(node6) or Follow(node5)      (d) (i)
First(node6)  =  {node6}                            (a)
Follow(node5) =  Follow(node2)                      (b) (ii)
Follow(node2) =  First(node2) or Follow(node1)      (d) (ii)
First(node2)  =  First(node3)                       (b) (i)
First(node3)  =  First(node4) or Follow(node3)      (d) (i)
          Doing First(node4) again - abandon
Follow(node1) =  {}                                 (f)
```

Figure 6.8. Degenerate case when applying `FIRST` and `FOLLOW`.

As can be seen, the situation arises where `FIRST(node4)` is about to be derived again. The reason this happens is because both `node4` and `node6` can follow `node4` but at the same time both `node4` and `node6` can also follow `node6`. This results in the implementation continually 'flipping' between `node4` and `node6`. Overcoming this problem is achieved simply by passing another parameter (not shown in Figures 6.6 and 6.7) with the four message patterns already discussed. This additional parameter stores the set of leaf nodes already traversed. Updating this new parameter is within the implementations of `first:` and `follow:` in class `Node`.

Guard Generation

Guards are produced during followset generation. To achieve this, another parameter is needed with the four message patterns described above. This parameter retains a concatenated string of conditions representing the guard for the leaf node whose followset is currently being generated. This string is compiled and realised as a method, and then installed in the method dictionary of the class representing the JSD process class being transformed. Guard generation is undertaken within two methods described earlier, namely:

```
nonLeafFirst:with:using:condition:
followFromChild:with:using:condition:
```

These are the complete message patterns for `nonLeafFirst:` and `followFromChild:with:`. Overleaf is an example of the full implementation of `followFromChild:with:using:condition:` in class `ItrNode`:

```
followFromChild: aNode with: aFollowMap using: testSet condition: condition
        "FOLLOW(Child) = FIRST(Child) or FOLLOW(Parent)"
        aNode
                first: aFollowMap
                using: testSet
                condition: (self buildCondition: condition
                        with: self firstSon condition).
        self
                follow: aFollowMap
                using: testSet
                condition: (self buildCondition: condition
                        with: self firstSon condition, ' not').
        ↑aFollowMap
```

As can be seen from this method, the second message pattern (beginning with `self`) builds the negative of the derived condition (by appending the string `not`), as explained in [5.3].

Guard Format

The format of guards in Smalltalk-80 is different to that discussed in [5.4]. Instead of having an individual and separate guard method for each state transition of the form `g(source, destination)`, all the guards relating to a particular destination are merged together into a single method. This results in each leaf node in the process structure having an associated compound guard method, its name being derived from the leaf node's name with the extension `G`. For example, the `Book` process would contain the seven compound guard methods:

`acquireG, classifyG, lendG, renewG, returnG, sellG, disposeG`

As an example, the internal structure of the compound guard method associated with leaf node `dispose` in the transformed `Book` process is as follows:

```
disposeG
        "g(classify, dispose)"
        self state = #classify ifTrue:
                [↑(self loanC not) and: [self disposeC]].
        "g(return, dispose)"
        self state = #return ifTrue:
                [↑(self loanC not) and: [self disposeC]]
```

`disposeG` reflects the fact that only the followsets `FOLLOW(classify)` and `FOLLOW(return)` contain amongst their entries the element `dispose`. Associated with these two followsets are the two guards `g(classify, dispose)` and `g(return, dispose)`, and hence these guards are implemented in the compound guard `disposeG`. Note that the conditionals are conjoined using the message `and:`. This message is known as a 'non-evaluating conjunction' — if the receiver of `and:` is the object `false`, then the parameter to `and:` is ignored and `false` is simply returned. Use of such messages speeds up guard evaluation considerably.

Followsets and guards have now been discussed with regard to their realisation in Smalltalk-80. However, in [5.4], it was shown that both followsets and guards are closely coupled together within the collection of dismembered parts after a process has been followmapped and inverted. The current

discussion has superficially separated the two from each other. The next section shows how the generation of followsets and guards is integrated.

6.3 Process Communication

Scheduling

So far, the description of the state vector separation and inversion transformations has concentrated solely on process specifications. In practice, however, the result of these two transformations on an actual JSD specification produces an architecture which is a hierarchy of subroutines. The root of this hierarchy is usually a special-purpose scheduler which realises all the necessary timing constraints which are part of a JSD specification. To reflect how the scheduling constraints are to be realised, the internal structure of this hierarchy can be altered by the developer.

There are three potential scheduling schemes available: *flat* and *knitting-needle* scheduling [Cameron86], and user-defined scheduling. All these schemes can optionally incorporate internal buffering of messages to help realise required timing constraints, although buffering is generally necessary when circuits of processes in the original specification exist. Briefly, flat scheduling produces a system where the hierarchy has only two levels, the first being the scheduler and below that, all the transformed processes. All communication between processes is effected via the scheduler; each process has been read or write-inverted, as appropriate, with respect to each of its datastreams. A knitting-needle scheduling scheme is generated basically by applying read inversion maximally throughout the JSD network. All external input is fed into the scheduler which then simply distributes those messages to their respective transformed processes. When a transformed process needs new data, it suspends and returns control to its caller, and possibly back up the hierarchy until control reaches the scheduler. An example of this scheme was given in [3.4] (Figure 3.7). As a further illustration, the two scheduling strategies are contrasted in Figure 6.9.

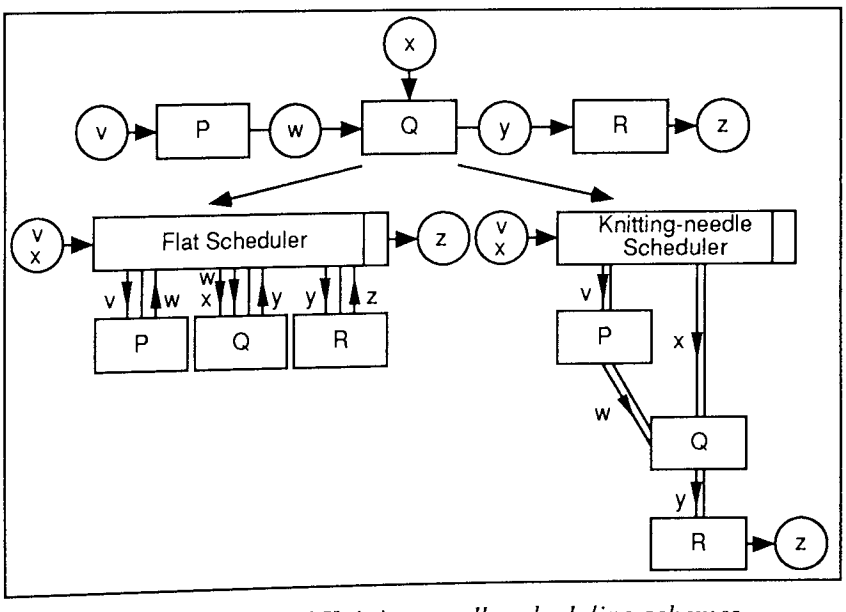


Figure 6.9. Flat and Knitting-needle scheduling schemes.

Although process destructuring in the Smalltalk-80 environment results in a collection of classes containing methods for leaf node operations, node conditions, guards and a collection of followsets, communication between instances of these classes is still via message sending and is synchronous. As was described in [5.5], read operations become assignments to a boolean flag, terminating the continuous scanning for the next dismembered part to be evaluated in the followmapped process. Write operations become calls to relevant transformed processes. The realisation in Smalltalk-80 is essentially no different. Basically, datastreams, which are instances of class `Datastream`, a subclass of the abstract class `JSDCommunication`, are stored as class variables of the process instance's class (for example as class variables of `BookProcess`) and are therefore global to all instances. The name of the datastream reflects the datastream connection's name in the network, e.g. the class variable `DS1` would represent the datastream connection called DS1 in a network. A `write:` message (or one of its derivatives) is used to resume a suspended process and gives the illusion that the writing of messages to datastreams in a JSD network has been carried through into the transformed system.

`Datastream` has two instances variables called `reader` and `writer` which store the names of the (transformed) producer and consumer processes respectively. Instances of class `Datastream`do not have any capability to buffer data, unlike their counterparts in JSD specifications. When the message `write:` is sent to an instance of a `Datastream`, the implementation first retrieves the name of the reader class. Since each transformed JSD process class holds onto all its instances via the class instance variable `instances`, the wanted instance is retrieved using its unique identification as a key. Once retrieved, the reading process 'wakes up'. This 'awakening' activity is effected by the *event manager*.

The event manager brings together both the followset structures (stored in the class instance variable called `followMap`) and compound guard methods to achieve behavioural congruence with executing JSD processes. The event manager is the generic name given to the `write:` method which animates a followmapped, inverted and state vector separated process in Smalltalk-80. The previous discussion in [5.4] showed how followsets and guards were distributed over many dismembered parts which themselves were integrated into a single procedure; this procedure continually looped round, evaluating the correct dismembered part until terminating at a suspend point. Below is the schema of the event manager plus a partial description of the related code:

❶ Using the process's current state (held in `QS`), access the relevant followset (found in the class instance variable `followMap`).

❷ For each general node state in the set, evaluate the associated guard functions until one returns `true`.

❸ Set the process's state to the new state.

❹ For the state which has a true guard, evaluate the code associated with that state.

❺ Repeat steps ❶ to ❹ until the process suspends (instance variable `suspended` becomes `true` — see overleaf).

```
anInstance resume.
[anInstance suspended]
    whileFalse: ❺
        [(process followMap at: anInstance state) ❶
            detect: [:element | anInstance perform: (element , 'G') asSymbol ❷ ]
            ifNone: [↑anInstance suspend].
        anInstance state: element. ❸
        anInstance perform: element ❹ ]
```

When a process instance wants to 'read' data from a 'datastream', the `read` message is sent to an instance of `Datastream` which simply redirects the read back to the transformed process instance which sent it and sets the `suspended` instance variable to `true`, thus terminating the execution of the event manager (this is exactly the same behaviour as described in [5.5]). In addition, the read mechanism saves in an instance variable of the transformed process instance the name of the datastream the process is about to suspend on. This enables suspended processes to be queried to determine which datastream they are waiting on — a capability needed by flat schedulers, for example.

The message interface of `write:` and `read` is necessary to guarantee that transformed processes behave as specified. Because it is possible for any method in a class to be invoked at any time, it is therefore possible to invoke methods relating to, say, general leaf node states in a transformed process class. Invoking methods such as these in any arbitrary order (other than that specified by a structure diagram) defeats the whole structuring constraint imposed, leading to transformed process instances being in incoherent states. Limiting all communication between transformed processes to the `write:` and `read` messages via the event manager ensures that process instances behave exactly as specified within network specifications.

State Vector Inspection

The implementation of state vector inspections is similar to datastream communications described above; the syntax used gives the illusion that the state vector connections within a network are actually being performed in the implementation. Instances of class `StateVector` (another subclass of `JSDCommunication`) are again stored in class variables of transformed JSD process classes. The class variable names are derived from the state vector inspections used in the network specification.

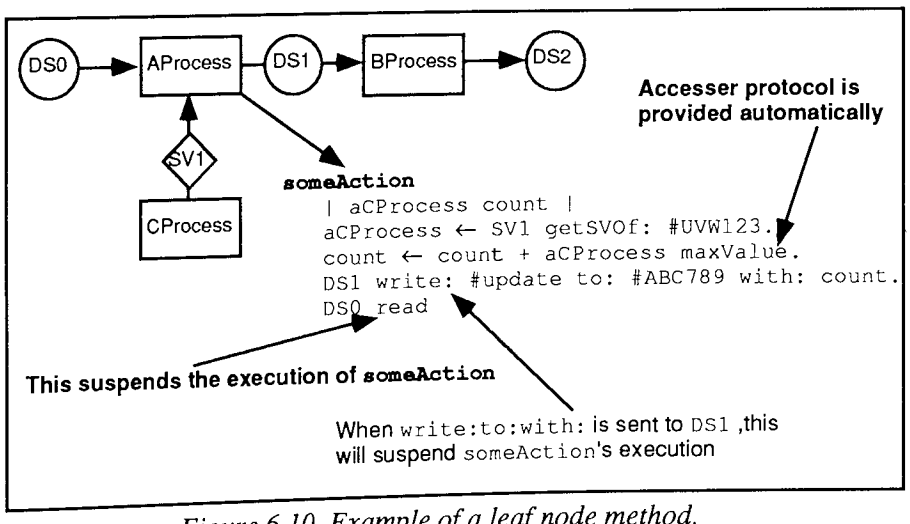


Figure 6.10. Example of a leaf node method.

`StateVector` has a single instance variable called `connection`, which stores the name of the transformed process class to be inspected. The two most important message protocol of `StateVector` are the messages `getSV` and `getSVOf:` These access the transformed JSD process class in `connection` and in turn access the dictionary of transformed process instances which each transformed JSD process class keeps. `getSV` returns all instances and `getSVOf:` returns a single specified instance. When individual parts of the state vector are required by the inspecting process, accesser protocol[2] is used to return values of individual instance variables, as illustrated in Figure 6.10 (on previous page).

Process Creation

One area, an essential part of the implementation stage in JSD, not yet covered is the actual creation of transformed process instances. Semantically, process instances in networks are considered to be 'eternal', neither being created nor deleted; specifications are deemed to have the correct number of processes to satisfy the requirements of the system[3]. This approach is entirely satisfactory when viewing the system at the abstract level of a specification, but of no use when attempting to implement that specification. This leaves process creation entirely at the discretion of the implementer. The approach adopted here is to create an instance of a transformed process class when the first `write:` message is sent to it; process creation is thus deferred as long as possible and is carried out 'on demand'.

6.4 The Process Browser

This last section gives a brief overview of the tool (also developed in Smalltalk-80) which supports the transformations described. The tool enables process structures to be entered graphically, the structures annotated with Smalltalk-80 code, and the code animated at any time during the structures' creation. The design of the tool itself is such that it does not need to be running in order to evaluate process specifications created by it. Figure 6.11 is a snapshot of the developed

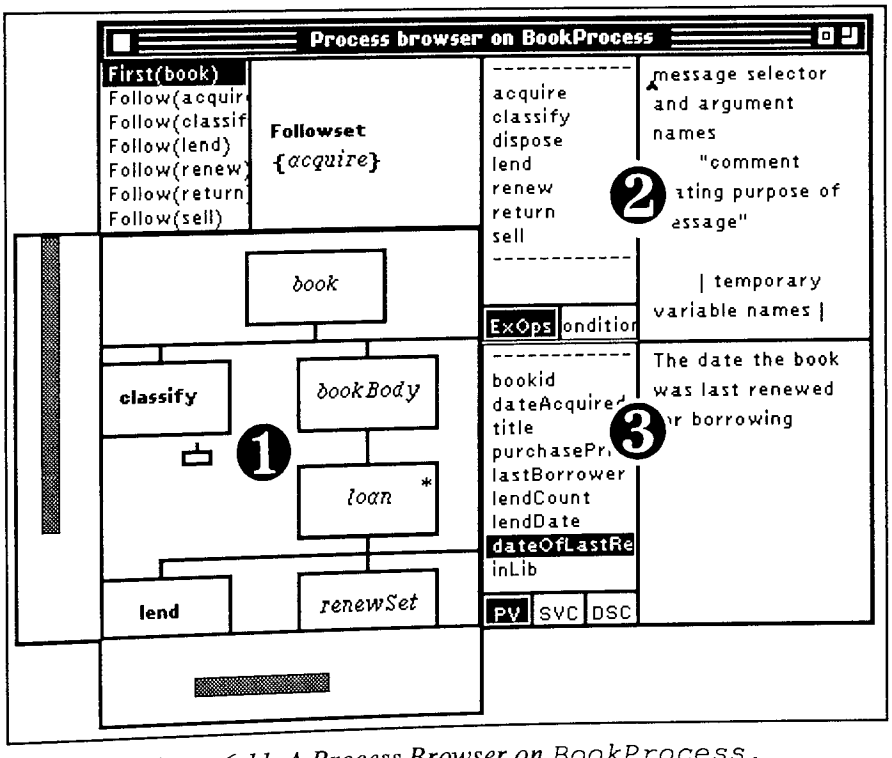


Figure 6.11. A Process Browser on `BookProcess`.

[2] Accesser protocol comprises those messages which assign and return values of instance variables.

[3] I am indebted to Michael Jackson for his explanation of this area.

process browser. It is composed of seven windows and two button clusters. The discussion of the browser will cover the three main areas labelled within it: structure editing, leaf node and conditional node operations, and finally process communication and process state definition.

Structure Editor

Features which have come to be expected in powerful, graphically driven software tools [Apple87] have been incorporated in the process browser and especially within the structure editor. As can be seen from Figure 6.10, diagrams can be scrolled around in two dimensional space, enabling the user to centre any node within the window and to manipulate it quickly. The structure editor enables users to build process structures by simply clicking on the mouse. As a visual feedback of what is going on, the cursor on the screen is context sensitive; i.e. the cursor changes its shape to reflect what is allowable depending on its position.

When the user brings the cursor near a displayed node in a diagram, the cursor changes shape to indicate the type of node addition that is allowed. Once the cursor has changed shape, the user simply clicks the mouse, and a new node will be added to the one being pointed at.

Figure 6.12 shows, amongst other things, a diagrammatic representation of an individual node (a node template) with its 10 different areas (numbered 1 to 10). When the cursor is over any one of these areas, the cursor changes its shape to indicate what can next be done when the mouse is clicked as mentioned above. For example, if the cursor is over area 8, then the cursor will change into a mini popup menu with the letters 'S' and 'V' next to it. This indicates that if the mouse is clicked, then a popup menu will appear relating to state vector inspections. Again, if the cursor is over area 4 then the cursor will change into a mini node indicating that when the mouse is clicked, a node will be added under the current node.

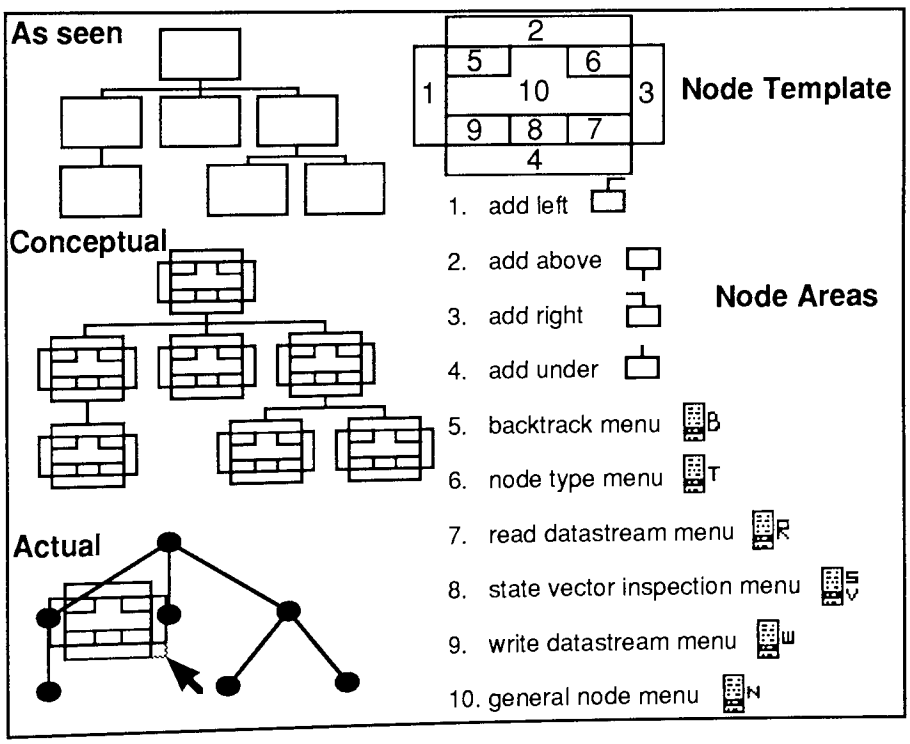


Figure 6.12. Node manipulation and visual feedback.

Conceptually, the way cursor context sensitivity is implemented is by having an instance of the node template for every node in a structure diagram as indicated in Figure 6.12. However, a more efficient

method of storing trees is simply to represent each node as a point, and attach (invisibly) an instance of the node template to the end of the cursor. When a point (representing a node) is under the now mobile node template and the mouse is clicked, it is merely a matter of doing a simple transformation on the coordinates of the cursor to find the area in which the node point is located in order to determine which activity can be carried out. The advantage of this approach is that, firstly, it reduces the amount of information needed to be stored, and secondly, all changes to the makeup of a node reside in one place — the node template. From this node template/point representation of a tree, a postscript representation of it can be created quite easily for high quality rendering on a laser printer.

Constraints built within the structure editor force a user to create diagrams which can only be structurally correct. These constraints are further enhanced by another aid within the structure editor which improves the aesthetics of the diagram as it is being created, namely that it is impossible to create an 'untidy' diagram. Each time a new node is added to a diagram (after the addition has satisfied the constraints) the diagram is automatically tidied. The tidying of structure diagrams is based on the tree tidying algorithm given by [Wetherell79].

One final point regarding the structure editor is the representation of a process in the form of a collection of followsets (only those associated with leaf nodes and the root node). This alternative representation is generated 'on the fly'. Whenever a new node is added to the diagram, the list of followsets (above the structure editor view) is automatically updated. Also, to make the display of these followsets more useful, the followset relating to the node in the structure diagram which currently has the cursor over it is automatically highlighted in the list. As already discussed, with each entry in a followset there is an associated compound guard. Although there are no views in the process editor showing guards, they are automatically generated and installed in the appropriate class. Direct editing of guards has not been provided as this could result in an inconsistent specification being formed; also, guards are an essential part of the mechanics supporting the transformational system and so should not be accessible anyway — compiler writers do not usually provide means of altering how a compiler compiles[4].

Operation and Condition Editor

The area labelled with the number ❷ in figure 6.11 is composed of the two windows and buttons dealing with entering operations and conditions associated with structure diagrams. The approach to code management adopted in the process browser is to make it as simple as possible for a user. This is achieved by guaranteeing that, although a process specification may be incomplete, it can never be inconsistent. When the user creates a new process structure, a class is automatically generated and installed in the Smalltalk-80 environment. As the user creates a structure diagram by simple mouse clicking, methods are generated for all leaf nodes in that structure automatically and installed in the newly created class. Although this may appear an inefficient approach, since a method will be created for every node in the tree (and then removed when the node is no longer a leaf node), it guarantees that there will

[4] Although the entire implementation of the Smalltalk-80 compiler is available to a user.

always be consistency between the diagrammatic specification and its attendant code. As well as creating methods for leaf nodes, methods are automatically created for all conditional nodes (iterations and selections) in the diagram. The default condition generated for these methods is to simply return the object `true`. For example, from Figure 6.11, the iterative node labelled `loan` will have the associated method

```
loanC
        ↑true
```

installed in the newly generated class. This default generated code can then easily be edited within the browser although it should be recalled that guards themselves are not user-accessible.

In order to switch between conditional node code and leaf node code the user simply clicks the buttons labelled 'Operations' and 'Conditions' within the browser. Leaf node code initially appears as a blank method (i.e. a method selector on its own), under which the user enters Smalltalk-80 code to describe what is to happen when a particular leaf node is to have its associated code evaluated. As previously stated, these methods are automatically removed whenever leaf nodes are removed from a structure diagram.

Communication and Process State Definition Editor

The final area (area ❸ in Figure 6.11) of the process browser to be discussed concerns defining datastream read and writes and state vector inspections within a process. There are two ways of entering this information. The first is to bring up the associated popup menu over the list area and select the `add` option. There the user will be prompted to enter in the name of the process being inspected (if the 'SV' button has been selected) or the name of the reading process or writing process (if the 'DSC' button has been selected). Once a new state vector connection or datastream connection has been created, the user can then refer to that named datastream in code associated with leaf node operations within the structure, since a new class variable will have been added to the appropriate class thus causing it to be in scope to all methods of the class.

An alternative approach to specifying datastreams and state vectors is to bring up the appropriate popup menu in the process browser over a particular leaf node. (Recall that the cursor changes its appearance indicating which popup menu will be displayed when the mouse is clicked over a leaf node). If a read datastream operation is wanted, then the system will automatically insert into the associated method (associated with a particular leaf node) the datastream read operation as the last statement of that method (see Figure 6.10 where the code `DS0 read` has been automatically appended to the method `someAction`). If the user removes that datastream from the process specification, then the system will automatically remove that read datastream from associated methods and recompile those that have been changed. Manipulating datastream reads is relatively simple as only one such operation per leaf node is permitted, and that must be the last operation. Writes to datastreams present a problem since there can be more than one in a leaf node and can appear anywhere within the code. Because of this, the user has to enter in the writes to datastreams manually in the code. This limitation which applies to writes to

datastreams, also applies to state vector inspections, which are also entered into the process specification in the same way.

Finally there is the 'PV' (Process Variables) button. This enables the user to enter in the state variables of the process specification. Once a new state variable is entered, the entire newly created class is recompiled, just as any class in the Smalltalk-80 system would have to be when a new instance variable is added to its specification. Once the class is recompiled, that process variable becomes in scope to all the methods of the class.

Detailed discussion of the process browser's design goes beyond the scope of this thesis. Suffice it to say that the browser has been designed to enable a user quickly to enter and evaluate valid process specifications. From the comments of those that have used it, this goal appears to have been satisfied.

Evaluating Generated Code

Once a user has completed specifying a process using the process browser, the generated Smalltalk-80 code representing the process can be evaluated. Since the code representing transformed processes are classes, they are simply installed in the Smalltalk-80 environment like all other classes and so are available for use. Figure 6.13 shows a Smalltalk-80 workspace in which there are several message patterns which can be evaluated (i.e. by selecting one and then choosing `doit` from a popup menu).

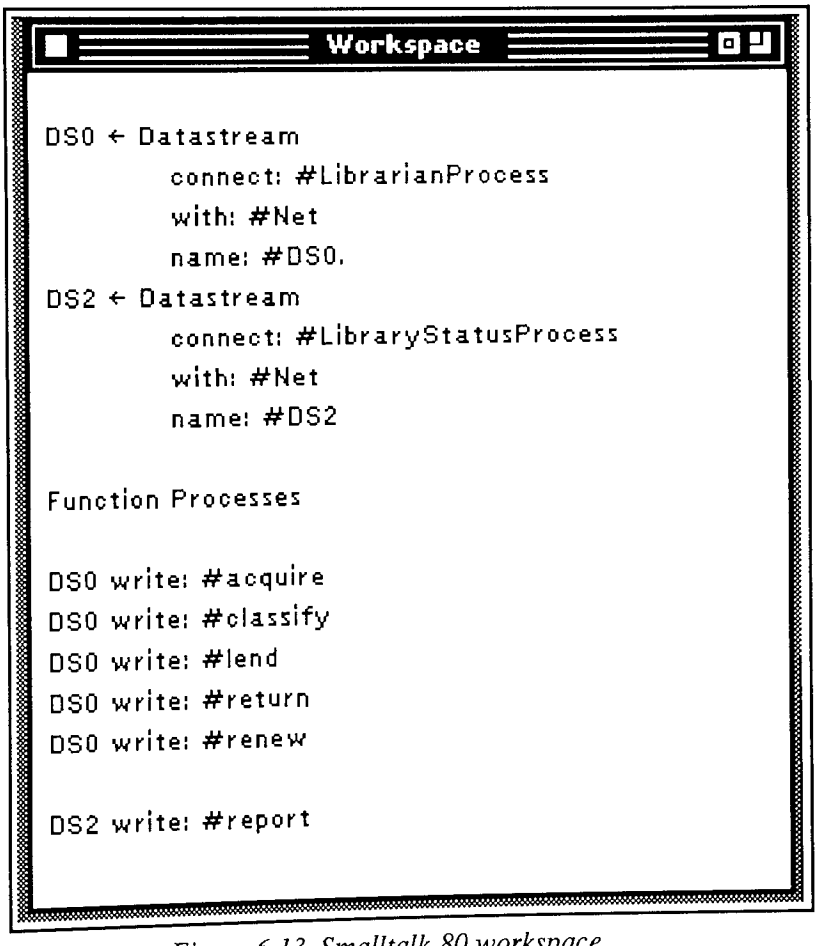


Figure 6.13. Smalltalk-80 workspace.

Since communication between the user and transformed processes is achieved via datastreams, in order for a user to see the behaviour of a specified process (or a network of them), the user will have to 'connect himself' to one or more processes of his choice. This is done by creating instances of `Datastream` and connecting them to the appropriate consumer process classes and the Smalltalk-80 environment (referred to as `#Net`) which effectively represents the 'outside real world'. All the user then has to do is evaluate the `write:` message expressions being sent to those datastream instances. For example, in Figure 6.13, `DS0 write: #return`.

In many cases, a process will start to run which in turn might send `write:` messages to other datastreams, thus causing further processes to be activated. However, little feedback is given to the user when this stimulus-response activity occurs. In order to overcome this, there is a facility in the process browser which allows the user to turn on a tracing facility. When the tracing facility has been enabled, all `write:` message sends are reported to a Smalltalk-80 system transcript indicating when the message send took place, which process resumed execution, and what state that resumed process is now in. Figure 6.14 shows a partial trace of evaluating `DSO write: #acquire` shown in Figure 6.13.

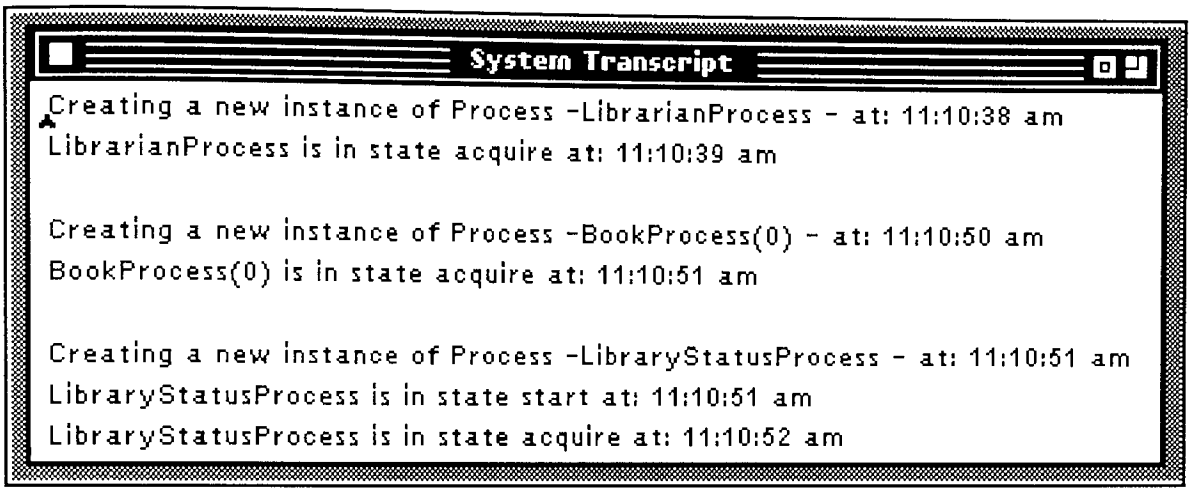


Figure 6.14. Trace of activity from a small network of transformed processes.

The trace information can, if needed, be exported to a file for future reference. Although the tracing mechanism is a little crude at present, it does give a user some visual feedback on how the specified system is behaving.

There will be those who will say that I am not qualified to undertake such a work and there will even be those who will say that I have no right to publish such things.

A.R. Butz

7 Conclusions

7.1 JSD and the Object Oriented Paradigm

Links between the two domains

The main thrust of this research has been in identifying a transformational path for implementing JSD specifications in object oriented languages and demonstrating those transformations in a tool. As well as fulfilling the original aim of the research, there have been some important byproducts which will now be discussed

Studying both the JSD method and the object oriented paradigm was the first major activity of this work. From that initial investigation, one conclusion which can be drawn is that JSD is not an object oriented design/development method. Qualifying this statement slightly, the activities of modelling the problem domain, the first phase of JSD, parallels closely the aims of an object oriented development method, and so the JSD modelling phase could legitimately be classed as object oriented. This observation concurs with others such as [Birchenough89]. However, as was discussed in Chapter 3, the two diverge from each other significantly after the modelling phase, as JSD proceeds to generate an abstract operational specification and eventually an implementation from that specification via transformation. Points of congruence do occur in these two phases between JSD and the object oriented paradigm as was discussed, but not to the extent that JSD can be justifiably classified as object oriented. The pigeon hole in which JSD found itself before the object oriented explosion was the operational paradigm. Although object oriented techniques are becoming increasingly popular, there is still no reason why JSD's original catergorisation should be changed — there is little doubt that JSD is currently the most successful example of an operational approach to software development, given the extent of its usage in the real world.

An important result of this initial study of the two domains was identification of pointers as to how best to map different aspects of JSD specifications into object oriented languages. For example, the exploitation of the class/instance mechanism found in the implementation of object oriented languages neatly realises the transformation of state vector separation. Other observations from that study have manifested themselves in the developed transformations.

To date, there has not been any major JSD development in which the final implementation has been realised in an object oriented programming language[1]. One of the main reasons for this is that the currently available tools supporting the method have only procedural code generators built within them. Some might counter this observation about lack of object oriented implementation by pointing out that certain JSD systems have been successfully built using Ada. Much work has been carried out in identifying appropriate transformations to map JSD specifications into Ada (see [Cameron89a]). However, Ada is not an object oriented programming language[2], which is also the opinion of others in the field such as [Cook86, Wegner89]. Booch, an authority on object oriented design and Ada, states that "Ada is distinctly not an object oriented programming language. Among other things, it lacks a mechanism for inheritance" [Booch89]. Wegner [Wegner89], like Booch, places Ada in the 'object-based' category of his (Wegner's) language classification hierarchy because it does not support inheritance or have classes. It is regarded as object-based because (according to Wegner) a package can be viewed as a form of object.

This research has described a transformational route for realising a JSD specification in any object oriented language and in doing so has extended the method's applicability. Further, the transformation of process destructuring which has been developed is sufficiently general to enable inversion to be applied to any procedural language. This is highlighted by the fact that JSD specifications can now be realised in languages such as Occam. In the case of Occam, there is no need to use the process primitive (PROC) provided in the language to implement all a specification's potential process instances (i.e. mapping a process instance to an Occam process); instead the approach described in this thesis can be used[3], producing a more efficient implementation (e.g. by avoiding dynamic sparsity). Although it would be valid to use the current transformational scheme to generate code in a hybrid object oriented programming language (such as C++ or Object Pascal), the best way to implement inversion in these languages is to simply use the procedural parts of the language. State vector separation, however, can be implemented in these hybrid languages by exploiting the class/instance mechanism as already described.

Other applications of the derived transformations, especially with regard to the use of followsets, include the ability to generate automatically context filter processes (see [5.1]), part of the input subsystem of JSD specifications. Also, handling structural problems such as recognition difficulties highlighted in [6.1], plus the generation of a finite state machine from a process structure, are possible uses of followsets. Some of these new areas of work are being investigated by [Bass92], and as such, this research has formed a partial stepping stone for Bass's work. Finally, one other area on which this research has had an influence is project PRESTIGE [Bass91]. More will be said later about this project but briefly, PRESTIGE is a workbench that enables a user to transform JSD specifications and automatically generate code in many procedural languages.

---

1 That is, not documented as of June 1991.

2 In the author's opinion.

3 Inversion in its standard realisation cannot be implemented in Occam and there is no 'goto' primitive.

7.2 *Paradigm Merge*

Alternative Life-Cycle

Since it is now possible to realise JSD specifications in, say, Smalltalk-80 it would be reasonable to ask who would want to implement a system in such a language? It has been shown that Smalltalk-80 is ideal for prototyping new and complex ideas very quickly [Diederich87]. This was the main reason why Smalltalk-80 was used for developing the process browser. However, Smalltalk-80 is seen as being too slow and insecure as a delivery platform due to it being an entirely dynamically bound and message-passing language (although some have countered this argument; see [LaLonde89]). These characteristics of Smalltalk-80 have prompted other researchers in trying to find an alternative delivery platform once a prototyping exercise in Smalltalk-80 is complete [Wirfs-Brock88]. Since a potential alternative development approach, described below, has emerged from this research, the original reason for choosing Smalltalk-80 to realise JSD specifications simply because it was 'the' object oriented programming environment can now be superseded by a better and more powerful argument in its favour.

Some years ago, the thought of using JSD as a prototyping medium would not have been considered — JSD was always viewed as a semi-formal development method, sitting at a different place in the spectrum of development methods compared to prototyping approaches. This thesis demonstrates that a number of different approaches to software construction can be integrated together to form the basis for a new paradigm for system development. This statement can be justified as follows. Initially, an operational specification is developed via JSD by entering process specifications into the process browser. Next transformations are automatically applied to these process specifications, generating Smalltalk-80 code and hence a runnable system. Since the turnaround time of seeing parts of a specification executed is small, it is reasonable to view the specification as a prototype [Feather82]. This ability to enter specifications and see their behaviour almost immediately offers a specification prototyping capability [Davis82], thus fully realising the operational nature of JSD. Hence, the development of a JSD specification can now be viewed as a form of prototypic 'implementation'. This view of specification development was commented on by Balzer [*cf.* Smoliar82] back in 1982: "If specifications are executable, then the original specification and all derived versions are programs, and all transformations are from programs to programs. Further, the executable specification can and should be used as a prototype. A prototype is then simply an incomplete implementation". Development of an 'incomplete implementation' (i.e. specification) which is executable but not very efficient, is now possible. Smalltalk-80 and the transformational research thus far developed have highlighted JSD's prototyping and executable capability, but have not addressed the problem of delivering a final implementation. This is where project PRESTIGE enters the scenario.

Project PRESTIGE has taken on board most of the practical results reported in this thesis. One feature PRESTIGE offers is the ability to adorn process specifications not with Smalltalk-80 but with a language called ESTEL. ESTEL is a procedural-like language, but its main strength lies in its ability to be transformed easily into most other procedural languages. This 'Esperanto' characteristic of ESTEL enables a specification to be implemented in, say, Ada, Pascal, C, or even Occam. Now a tool set supporting JSD using ESTEL would be a powerful tool in its own right, but what gives PRESTIGE an even greater flexibility is that ESTEL can also be transformed into Smalltalk-80. Therefore, all the prototyping capabilities already referred to are now available, plus the ability to generate a final system in, say, Ada. Hence, PRESTIGE alters the life-cycle in which JSD originally found itself. One view of PRESTIGE's capability is that the user is now able to experiment at the specification level, and not, as with any other tools, with the code which is generated by them. This gives the user a certain leverage over and above what is offered by most other tools currently supporting the JSD method, and for that matter many other development methods.

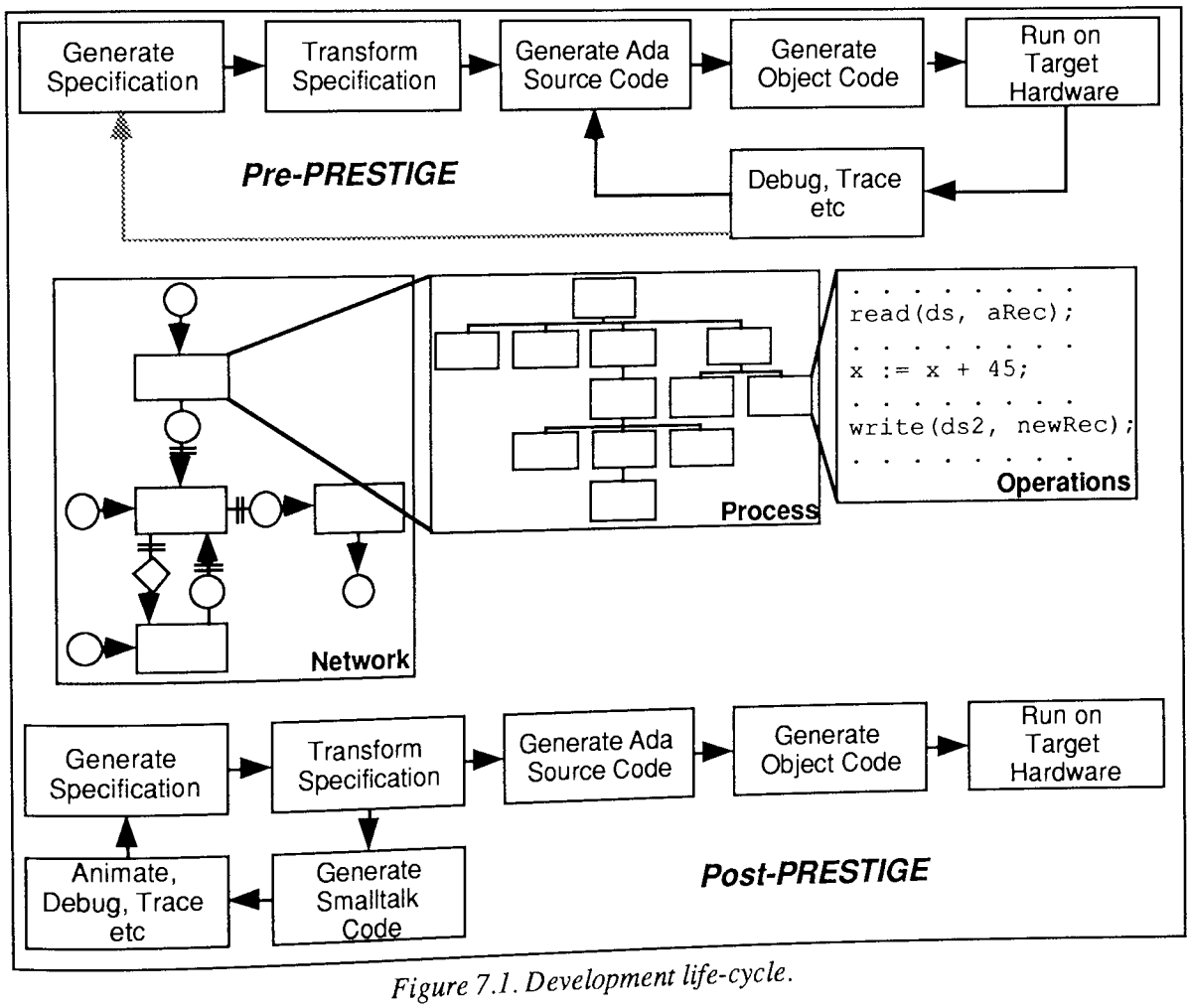


Figure 7.1. Development life-cycle.

Figure 7.1 attempts to show the impact of project PRESTIGE on the software development process. Before PRESTIGE, the tools which supported JSD enabled a user to develop a specification, and then

transform that specification into source code largely by non-automated means (in the diagram the language Ada is used as an example). The generated code would be exported from the tool set. The next stage in the development process would be to compile the code and then run it. Any problems in the behaviour of the running system would be resolved by editing the generated source code. This is indicated in the top part of the diagram, where the standard edit-compile-run-debug loop would be continually iterated through.

With the advent of the PRESTIGE workbench and its exploitation of this research, this development life-cycle can be improved. A JSD specification can now be transformed into Smalltalk-80. The transformed specification can then be run within the Smalltalk-80 environment, albeit more slowly than would otherwise be the case in a procedural language. Behavioural problems can now be intercepted at this prototyping stage, and the specification modified. When the user is satisfied that the system behaves correctly, the specification can be transformed into another language and have much greater confidence that the newly generated system will behave correctly. The PRESTIGE development path ensures that the specification is always consistent with the implementation, since all development is carried out with the tool (except for the compilation of the generated code). Before the PRESTIGE workbench, the implementation could easily become inconsistent with the specification, since developers would tend to alter the generated code not via the specification tool but directly. PRESTIGE, coupled with the results of this research, has opened up a new and potentially improved pathway for deliverable systems.

7.3 *Known Deficiencies and Future Developments*

Transformations

Applying inversion to a destructured process relies on the fact that each dismembered part has no internal structure i.e. no iterations or selections. However, this necessary characteristic of the transformation process precludes a class of solutions which currently can be supported. The class of solutions are those which use backtracking (see [6.1]). Backtracking would introduce additional structure within dismembered parts by allowing control to switch from one dismembered part to another. At present, rules for generating followsets from process structures which contain backtracking have not been developed. Although this is a drawback, it does not completely restrict process specification since processes which would benefit from the use of backtracking can be specified by using multiple nested selections instead — an inelegant solution, but one which works.

Probably the most serious drawback is that a process structure cannot have two nodes with the same name. At present, this rules out some potential process structures. However, a relatively simple way to overcome this defect would be to assign a unique index to every node in a process structure, and using the actual node's name as an alias. The remedying of this defect together with that mentioned above concerning backtracking would enable the totality of process specifications to be encompassed.

To make the tool more comprehensive, the integration of context manipulation and followset generation would provide even greater flexibility in process specification. Context manipulation (see [4.4]) is the transformational path employed when realising ESTEL code in Smalltalk-80. Context manipulation is more efficient than a followset-based approach. However, followsets have a greater potential in terms of being applied to a variety of application areas, as already discussed. A possible optimisation, given that in Smalltalk-80 one can add user-defined primitives to the virtual machine, would be to make the `suspend` and `resume` methods in `ReEntrantObject` primitive. This would undoubtedly increase the performance of the execution of `suspend` and `resume` but at the expense of making the generated code which uses these methods unportable (since every virtual machine would have to be modified in order to run the primitive methods).

As regards the process browser itself, this is still not 'correct' from an aesthetic point of view, or from a functional point of view. There should exist an independent node editor for entering in executable operations and conditions. The current portion of the browser where executable operations are entered should be the place where individual (leaf and root) node documentation is dealt with. From a model process view point, leaf node documentation would equate to documenting the actions which make up a process. At present there is no direct way to document a process itself — an important aspect which has been overlooked. Documenting the root node would alleviate this problem. Another problem which tends to arise is the size of process specifications themselves. When the diagram gets to a certain size, it is possible at present to scroll it around in two-dimensional space. However, if the diagram becomes very large, merely being able to scroll it around is not enough. What is needed is the ability to hide or 'fold' away different parts of the diagram, and unfold them when needed. Implementing this feature is not seen as a difficult task and would give the tool some additional 'polish' when being used.

In conclusion, the byproducts and offshoots of various research endeavours sometimes have greater impact than the original research itself. This piece of research hopefully will be seen to fall into this category. As well as being able to transform JSD specifications into any object oriented language, it has been shown that JSD specifications can now be prototyped and transformed very quickly into runnable systems. Cameron's belief that "a JSD specification is ... directly executable, at least in principle" [Cameron86], can justifiably be claimed to have now been realised, although perhaps not in the way he was originally envisaging. It is this author's belief that, now that the practical results of this piece of research have been incorporated into PRESTIGE, and with more work on improving the capabilities of the workbench, eventually evolving it into an industrial-strength tool, the full potential of JSD as a powerful development method will have been realised.

References

Agha86 — Agha G., *Actors: A Model of Concurrent Computation.* MIT Press, 1986.

Agresti86 — Agresti W.W., 'What are the new Paradigms?'. In *New Paradigms for Software Development,* (Ed. W.W. Agresti), IEEE Computer Society Press, pp. 6-10, 1986.

Aho85 — Aho A.V., Sethi R. & Ullman J.D., *Compilers Principles, Tools and Techniques.* Addison-Wesley, 1985.

America86 — America P., 'Object-oriented programming: a theoretician's introduction', *Bulletin European Association for Theoretical Computer Science (Austria),* no. 29, pp. 69-84, June 1986.

America87 — America P., 'Inheritance and Subtyping in a Parallel Object-Oriented Language'. In *ECOOP '87, Proceedings of European Conference on Object-Oriented Programming* (Eds. J. Bézivin, J.-M. Hullot, P. Cointe & H. Lieberman), Paris, France, 15-17 June. *Lecture Notes in Computer Science,* no. 276, pp. 234-242, Springer-Verlag 1987.

Andrews87 — Andrews T. & Harris C., 'Combining Language and Database Advances in an Object-Oriented Development Environment'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, Orlando, Florida, USA, 4-8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 430-440, Dec. 1987.

Apple87 — Apple Computer Corp., *Human Interface Guidelines: The Apple Desktop Interface.* Addison-Wesley 1987.

Apple90 — Apple Computer Corp., *MacApp 2.0. General Reference.* Developer Technical Publications, Apple Computer Corp. 1990.

Aretz82 — Aretz F.E.J.K., 'Abstraction', *Phillips Technical Review,* **40**(8/9) pp. 225-229 1982.

Atkinson83 — Atkinson M.P., Bailey P.J., Chisholm K.J., Cockshott P.W. & Morrison R., 'An Approach to Persistent Programming', *The Computer Journal (GB),* **26**(4), pp. 360-365, 1983.

Atkinson85 — Atkinson M.P & Morrison R., 'Procedures as Persistent Data Objects', *ACM Transactions on Programming Languages and Systems (USA),* 7(4) pp. 539-559, Oct. 1985.

Atkinson86 — Atkinson M.P. & Buneman O.P., 'Types and Persistence in Database Programming Languages', *ACM Computing Surveys (USA),* **19**(2), pp. 105-190, June 1987.

Atkinson88 — Atkinson M.P., Buneman O.P. & Morrison R., 'Binding and Type Checking in Database Programming Languages', *The Computer Journal (GB),* **31**(2), pp. 99-109, April 1988.

Baden84 — Baden S.B., 'Low-Overhead Storage Reclamation in the Smalltalk-80 Virtual Machine'. In *Smalltalk-80 Bits of History, Words of Advice,* (Ed. G. Krasner), pp. 331-342, Addison-Wesley, 1984.

Bailin89 — Bailin S.C., 'An object oriented requirements specification method', *Communications of the ACM (USA)*, **32**(5), pp. 608-23, May 1989.

Baines89 — Baines R., 'Object oriented programming', *Electronic and Wireless World (GB)*, **95**(1638), pp. 370-4, April 1989.

Balzer81 — Balzer R.M. & Cheatham T.E., 'Editorial: Program Transformations', *IEEE Transactions on Software Engineering (USA)*, **SE-7**(1), pp. 1-2, Jan. 1981.

Balzer82 — Balzer R.M., Goldman N.M. & Wile D.S., 'Operational Specification as the basis for Rapid Prototyping', *ACM SIGSOFT Software Engineering Notes (USA)*, 7(5), pp. 3-16, Dec. 1982.

Bass90 — Bass A., 'Follow-set based transformations for the implementation of JSD specifications'. Project PRESTIGE, Dept. Computer Science, Aston University, UK. (Document EXWP//9(1.0)/310390), 1989.

Bass91 — Bass A., Boyle M. & Ratcliff B. 'PRESTIGE: A CASE workbench for the JSD implementor'. *In 13th International Conference on Software Engineering*, Austin, Texas. IEEE Computer Society Press, pp. 198-207, 1991.

Bass92 — Bass A. Forthcoming PhD Thesis, Aston University, UK, 1992.

Beck89 — Beck K., Program Introduction. In *OOPSLA '89. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, New Orleans, Louisiana, USA, 1-6 Oct. 1989 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **24**(10), p. v, Oct. 1989.

Birchenough89 — Birchenough A. & Cameron J.R., 'JSD and Object Oriented Design'. In *JSP & JSD: The Jackson Approach to Software Development*, (Ed. J.R. Cameron), IEEE Computer Society Press, 2nd Edition, pp. 293-304, 1989.

Birchenough89a — Birchenough A. & Cameron J.R., 'The Implementation of LSD specifications via JSD'. ARE(MOD) internal report, Dec. 1989.

Bobrow83 — Bobrow D.G. & Stefik M., *The LOOPS Manual.* Xerox Corporation, 1983.

Booch83 — Booch G., *Software Engineering with Ada.* Benjamin/Cummings (1983).

Booch86 — Booch G., 'Object-Oriented Development', *IEEE Transactions on Software Engineering (USA)*, **SE-12**(2), pp. 211-221, Feb. 1986.

Booch89 — Booch G., 'Position Paper on using Ada with Object-Oriented Design'. In *OOPSLA '89. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, New Orleans, Louisiana, USA, 1-6 Oct. 1989 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **24**(10), p. 494, Oct. 1989.

Borgers90 — Borgers M. & Munro M., 'Producing Better Maintainable JSD Specifications by Grouping Common Aspects', *Software Maintenance: Research and Practice (GB)*, **2**(1), pp. 61-80, March 1990.

Borning87 — Borning A., 'Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language'. In *ECOOP '87, Proceedings of European Conference on Object-Oriented Programming* (Eds. J. Bézivin, J.-M. Hullot, P. Cointe & H. Lieberman), Paris, France, 15-17 June. *Lecture Notes in Computer Science*, no. 276, pp. 1-10, Springer-Verlag 1987.

Brown90 — Brown J., 'Is the Universe a Computer', *New Scientist*, no. 1725, pp. 37-39, 14 July 1990.

Byte81 — *BYTE (USA)*, **6**(8), Aug. 1981. Special issue on Smalltalk-80.

Cameron83 — Cameron J.R. (Ed), *JSP & JSD: The Jackson Approach to Software Development.* IEEE Computer Society Press, 1983.

Cameron86 — Cameron J.R., 'An Overview of JSD', *IEEE Transactions on Software Engineering (USA)*, **SE-12**(2), pp. 222-240, Feb. 1986.

Cameron88 — Cameron J.R., 'The modelling phase of JSD', *Information and Software Technology (GB)*, **30**(6), pp. 373-383, July/August 1988.

Cameron89 — Cameron J.R. (Ed), *JSP & JSD: The Jackson Approach to Software Development.* IEEE Computer Society Press, 2nd Edition, 1989.

Cameron89a — Cameron J.R., 'Mapping JSD Network Specifications into Ada'. In *JSP & JSD: The Jackson Approach to Software Development*, (Ed. J.R. Cameron), pp. 305-313, IEEE Computer Society Press, 2nd Edition, 1989.

Cardelli85 — Cardelli L. & Wegner P., 'On Understanding Types, Data Abstraction, and Polymorphism', *ACM Computing Surveys (USA)*, **17**(4), pp. 471-522, Dec. 1985.

Clocksin81 — Clocksin W.F. & Mellish C.S., *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

Cockshott84 — Cockshott W.P., Atkinson M.P., Bailey P.J., Chisholm K.J. & Morrison R., 'Persistent object management system', *Software-Practice and Experience (GB)* **14**(1), pp. 49-71, Jan. 1984.

Cohen81 — Cohen J., 'Garbage Collection of Linked Data Structures', *ACM Computing Surveys (USA)*, **13**(3), pp. 341-367, Sept. 1981.

Cohen82 — Cohen D., Swartout W. & Balzer R., 'Using Symbolic Execution To Characterize Behavior', *ACM SIGSOFT Software Engineering Notes (USA)*, **7**(5), pp. 25-32, Dec. 1982.

Cointe87 — Cointe P., 'Metaclasses are First Class: the ObjVlisp Model'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, Orlando, Florida, USA, 4-8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 156-167, Dec. 1987.

Cook86 — Cook S., 'Languages and object-oriented programming', *Software Engineering Journal (GB)*, **1**(2), pp. 73-80, March 1986.

Cooper90 — Cooper S., 'The double O approach', *Computers in Defence (GB)*, **4**(1), pp. 22-24, Jan/Feb. 1990.

Cox86 — Cox B.J., *Object-Oriented Programming, An Evolutionary Approach.* Addison-Wesley, 1986.

Cox87 — Cox B.J. & Schmucker K.J., 'Producer: A Tool for Translating Smalltalk-80 to Objective-C'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, Orlando, Florida, USA, 4-8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 423-429, Dec. 1987.

Dahl66 — Dahl O. & Nygaard K., 'Simula - An Algol based Simulation Language', *Communications of the ACM (USA)*, **9**(9) pp. 671-678, Sept. 1966.

Danforth88 — Danforth S. & Tomlinson C., 'Type theories and object-oriented programming', *ACM Computing Surveys (USA)*, **20**(1), pp. 29-72, March 1988.

Davis82 — Davis A.M., 'Rapid Prototyping using Executable Requirements Specifications', *ACM SIGSOFT Software Engineering Notes (USA)*, **7**(5), pp. 39-44, Dec. 1982.

Deutsch76 — Deutsch L.P. & Bobrow D.G., 'An efficient incremental automatic garbage collector', *Communications of the ACM (USA)*, **19**(9), pp. 522-526, Sept. 1976.

Deutsch84 — Deutsch L.P. & Schiffmann A.M., 'Efficient Implementation of the Smalltalk-80 System'. In *ACM SIGACT/SIGPLAN Proceedings of the Eleventh Annual Symposium on the Principles of Programming Languages*, Salt Lake City, Utha, USA. pp. 297-302, 1984.

Deutsch89 — Deutsch L.P., 'The Past, Present and Future of Smalltalk'. In *ECOOP '89, Proceeding of the third European Conference on Object-Oriented Programming*, (Ed. S. Cook), Nottingham, UK, 10-14 July. Cambridge University Press, pp. 73-87, 1989.

Diederich87 — Diederich J. & Milton J., 'Experimental prototyping in Smalltalk', *IEEE Software (USA)*, **4**(3), pp. 50-64, May 1987.

Duhl88 — Duhl J. & Damon C., 'A Performance Comparison of Object and Relational Databases Using the Sun Benchmark'. In *OOPSLA '88. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, San Diego, California, USA, 25 Sept. - 30 Sept. 1988 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **23**(11), pp. 153-163, Nov. 1988.

Feather82 — Feather M.S., 'Mappings for Rapid Prototyping', *ACM SIGSOFT Software Engineering Notes (USA)*, **7**(5), pp. 17-24, Dec. 1982.

Fikes85 — Fikes R. & Kehler T., 'The Role of Frame-Based Representation in Reasoning', *Communications of the ACM (USA)*, **28**(9), pp. 904-20, Sept. 1985.

Florentin85 — Florentin J.J., 'New constructs in programming languages', *Computer Bulletin (UK)*, **1**(3/2), pp. 10-13, June 1985.

Foote89 — Foote B. & Johnson R.E., 'Reflective Facilities in Smalltalk-80'. In *OOPSLA '89. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, New Orleans, Louisiana, USA, 1-6 Oct. 1989 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **24**(10), pp. 327-335, Oct. 1989.

Friedman84 — Friedman D.P., Haynes C.T. & Kohlbecker E., 'Programming with Continuations'. In *Program Transformation and Programming Environments*. (Ed. P. Pepper), NATO ASI series, Springer-Verlag Berlin, vol. F8, pp. 263-274, 1984.

Goldberg83 — Goldberg A. & Robson D., *Smalltalk-80: The language and its implementation.* Addison-Wesley 1986.

Goossenaerts88 — Goossenaerts J. & Lewi J., 'Object-Oriented Programming: Concepts'. In *CompEuro88 — System Design, Concepts, Methods and Tools*. IEEE Computer Society Press, pp. 2-8, 1988.

Guttag77 — Guttag J., 'Abstract Data Types and the Development of Data Structures', *Communications of the ACM (USA)*, **20**(6), pp. 396-404, June 1977.

Harland84 — Harland D.M., *Polymorphic Programming Languages — design and implementation.* Ellis Horwood, 1984.

Harland84a — Harland D.M. & Gunn H.I.E., 'Polymorphic Programming I. Another Language designed on Semantic Principles', *Software-Practice and Experience (GB)*, **14**(10), pp. 973-997, Oct. 1984.

Harland88 — Harland D.M., *REKURSIV: Object-Oriented Computer Architecture.* Ellis Horwood 1988.

Haynes86 — Haynes C.T., Friedman D.P. & Wand M., 'Obtaining Coroutines with Continuations', *Computer Languages*, **11**(3/4), pp. 143-153, 1986.

Haynes87 — Haynes C.T. & Friedman D.P., 'Embedding Continuations in Procedural Objects', *ACM Transactions on Programming Languages and Systems (USA)*, **9**(4), pp. 582-598, Oct. 1987.

Henderson86 — Henderson P., 'Functional Programming, Formal Specification, and Rapid Prototyping' *IEEE Transactions on Software Engineering (USA)*, **SE-12**(2), pp. 241-250, Feb. 1986.

Hendler86 — Hendler J., 'Enhancement for multiple-inheritance'. In Special issue of the ACM SIGPLAN notices on the Object-Oriented Programming Workshop held at IBM Yorktown Heights, June 9-13, 1986. *ACM SIGPLAN Notices (USA)*, **21**(10), pp. 98-106, Oct. 1986.

Hewitt77 — Hewitt C., 'Viewing Control Structures as Patterns of Passing Messages', *Artificial Intelligence (USA)*, **8**(3), pp. 323-364, June 1977.

Hopkins89 — Hopkins T.P. & Wolczko M.I. 'Writing Concurrent Object-Oriented Programs using Smalltalk-80', *The Computer Journal (GB)*, **32**(4), pp. 341-350, 1989.

Hughes79 — Hughes J.W., 'A Formalization and Explication of the Michael Jackson Method of Program Design', *Software-Practice and Experience (GB)*, **9**(3), pp. 191-202, 1979.

Hull84 — Hull M.E.C. & McKeag R.M., 'Concurrency in the Design of Data Processing Systems', *The Computer Journal (GB)*, **27**(4), pp. 289-293, 1984.

Ingalls78 Ingalls D.H.H., 'The Smalltalk-76 Programming System Design and Implementation'. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, ACM Press, New York, USA, pp. 9-16, 1978.

Ingalls81 Ingalls D.H.H., 'Design principles behind Smalltalk', *BYTE (USA)*, **6**(8), pp. 286-98, Aug. 1981.

Ingalls86 Ingalls D.H.H., 'A Simple Technique for Handling Multiple Polymorphism'. In *OOPSLA '86. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Portland, Oregon, USA, 29 Sept. - 2 Oct. 1986 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **21**(11), pp. 347-349, Nov. 1986.

Jackson75 Jackson M.A., *Principles of Program Design*. Academic Press, 1975.

Jackson80 Jackson M.A., 'Information systems: modelling, sequencing and transformations'. In *On the construction of programs*, (Ed. R.M. Macnaghten & A.M. McKeag), Cambridge, England, pp. 319-341, 1980. (Reprinted from Proceedings of Third International Conference on Software Engineering, May 10-12, 1978).

Jackson81 Jackson M.A., 'Some Principles Underlying a System Development Method'. In *Systems Analysis and Design - A Foundation for the 1980s*, (Eds W.W. Cotterman et al), New York: Elsevier North-Holland, 1981.

Jackson82 Jackson M.A., 'Software Development as an Engineering Problem', *Angewandte Informatik*, no. 2, pp. 96-103, Feb. 1982.

Jackson83 Jackson M.A., *System Development*. Prentice-Hall, 1983.

Jackson84 Jackson M.A., 'Structure-oriented programming'. In *Programming Transformation and Programming Environments*, NATO ASI Series, vol. F8, (Ed. P. Pepper), pp. 181-198, Springer-Verlag, 1984.

Jackson88 Jackson M.A., 'Objects and other Subjects'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications*. Addendum to the Conference Proceedings, Orlando, Florida, USA, 4-8 Oct. 1987 (Eds. L. Power & Z. Weiss), *ACM SIGPLAN Notices (USA)* **23**(5), pp. 97-104, May. 1988.

Jones90 Jones C.B., *Systematic Software Development Using VDM*. Second Edition, Prentice-Hall, Englewood-Cliffs, N.J. 1986.

Kafura89 Kafura D.G. & Lee K.H., 'Inheritance in Actor Based Concurrent Object-Oriented Languages'. In *ECOOP '89, Proceeding of the third European Conference on Object Oriented Programming*, (Ed. S. Cook), Nottingham, UK, 10-14 July. Cambridge University Press, pp. 131-145, 1989.

Kato87 Kato J. & Morisawa Y., 'Direct Execution of a JSD Specification'. In *Proceedings of Compsac 87, 11th Annual International Computer Software and Applications Conference*, Tokyo, Japan. pp. 30-37, 7-9 Oct. 1987.

Kay69 Kay A.C., *The Reactive Engine*. PhD Thesis, University of Utah, 1969.

Kay87 Kay A.C., 'Alan Kay: face values', *Personal Computer Weekly (GB)*, pp. 128-135, Dec. 1987.

Keene89 — Keene S.E., *Object Oriented Programming in Common Lisp - A programmers guide to CLOS*. Addison-Wesley 1989.

Kersten86 — Kersten M.L. & Schippers F.H., 'Towards an Object-Centered Database Language'. In *Proceedings of the 1986 International Workshop on Object-oriented Database Systems*. Washington, DC, USA. IEEE Computer Society Press, pp. 104-111, 1986.

Khoshafian86 — Khoshafian S.N. & Copeland G.P., 'Object identity'. In *OOPSLA '86. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Portland, Oregon, USA, 29 Sept. - 2 Oct. 1986 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **21**(11), pp. 406-16, Nov. 1986.

Knudsen88 — Knudsen J.L. & Madsen O.L., 'Teaching Object-Oriented Programming is more than teaching Object-Oriented Programming Languages'. In *ECOOP '88, Proceedings of the Second European Conference on Object-Oriented Programming* (Eds. S. Gjessing & K. Nygaard), Oslo, Norway, 15-17 Aug. *Lecture Notes in Computer Science*, no. 322, pp. 21-40, Springer-Verlag 1988.

LaLonde88 — LaLonde W.R. & Gulik M.V., 'Building a Backtracking Facility in Smalltalk Without Kernel Support'. In *OOPSLA '88. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, San Diego, California, USA, 25 Sept. - 30 Sept. 1988 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **23**(11), pp. 105-122, Nov. 1988.

LaLonde89 — LaLonde W.R., McGugan J. & Thomas D., 'The real advantages of pure object-oriented systems or Why object-oriented extensions to C are doomed to fail'. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, Orlando, Florida, USA, 20-22 Sept. 1989 (Washington, DC, USA: IEEE Comput. Soc. Press 1989), pp. 344-50.

Lawler82 — Lawler R.W., 'Designing Computer-Based Microworlds', *BYTE (USA)*, **7**(8) pp. 138-160, Aug. 1982.

Ledbetter85 — Ledbetter L. & Cox B.J., 'Software-ICs', *BYTE (USA)*, **10**(6), pp. 307-16, June 1985.

Lewis90 — Lewis C.T. & Ratcliff B., 'The Realisation of JSD Specifications in Smalltalk-80'. In *NATO DRG Panel 2, Information Processing Technology Symposium*, RSRE Malvern UK, May 8-9, 1990.

Lieberman86 — Lieberman H., 'Using Prototypical Objects to Implement Shared behavior in Object-Oriented Systems'. In *OOPSLA '86. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Portland, Oregon, USA, 29 Sept. - 2 Oct. 1986 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **21**(11), pp. 214-223, Nov. 1986.

Lieberman87 — Lieberman H., 'Concurrent Object-Oriented Programming in Act1'. In *Object-Oriented Concurrent Programming* (Eds. A. Yonezawa & M. Tokoro), pp. 9-36, MIT Press, 1987.

Liskov80 — Liskov B., 'CLU reference manual', *Lecture notes in Computer Science*, no. 114, Springer-Verlag, 1980.

Low88 — Low C., 'A Shared, Persistent Object Store', In *ECOOP '88, Proceedings of the Second European Conference on Object-Oriented Programming* (Eds. S. Gjessing & K. Nygaard), Oslo, Norway, 15-17 Aug. *Lecture Notes in Computer Science*, no. 322, pp. 390-410, Springer-Verlag 1988.

MacLennan82 — MacLennan B.J., 'Values and objects in programming languages', *ACM SIGPLAN Notices (USA)*, **17**(12), pp. 70-79, Dec. 1982.

Madsen88 — Madsen O.L. & Moller-Pedersen B., 'What object-oriented programming may be - and what it does not have to be'. In *ECOOP '88, Proceedings of the Second European Conference on Object-Oriented Programming* (Eds. S. Gjessing & K. Nygaard), Oslo, Norway, 15-17 Aug. *Lecture Notes in Computer Science*, no. 322, pp. 1-20, Springer-Verlag 1988.

Maes87 — Maes P., 'Concepts and Experiments in Computational Reflection'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Orlando, Florida, USA, 4-8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 147-155, Dec. 1987.

Maes87a — Maes P., *Computational Reflection*. PhD Thesis, Vrije Universiteit Brussel, 1987.

Maier86 — Maier D., Stein J., Otis A. & Purdy A., 'Development of an object-oriented DBMS'. In *OOPSLA '86. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Portland, Oregon, USA, 29 Sept. - 2 Oct. 1986, (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **21**(11), pp. 472-82, Nov. 1986.

Masiero88 — Masiero P.C. & Germano F.S.R., 'JSD as an Object Oriented Design Method', *ACM SIGSOFT Software Engineering Notes (USA)*, **13**(3), pp. 22-23, July 1988.

McCullough87 — McCullough P.L., 'Transparent Forwarding: First Steps'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Orlando, Florida, USA, 4-8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 331-341, Dec. 1987.

McIntyre88 — McIntyre S.C. & Higgins L.F., 'Object oriented systems analysis and design: methodology and application', *Journal of Management and Information Systems (USA)*, **5**(1), pp. 25-35, Summer 1988.

McNeile89 — McNeile A.T. & Powell J., 'Executable Specifications as a Tool for Systems Architecture'. In *JSP & JSD: The Jackson Approach to Software Development*, (Ed. J.R. Cameron), IEEE Computer Society Press, 2nd Edition, pp. 373-378, 1989.

Merrow87 — Merrow T. & Laursen J., 'A Pragmatic System for Shared Persistent Objects'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Orlando, Florida, USA, 4 - 8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 103-110, Dec. 1987.

Meyer87 — Meyer B., 'Reusability: the case for object-oriented design', *IEEE Software (USA)*, **4**(2), pp. 50-64, March 1987.

Meyer88 — Meyer B., *Object-Oriented Software Construction*, Prentice Hall, 1988.

Minkowitz87 — Minkowitz C. & Henderson P., 'A formal description of object-oriented programming using VDM'. In *VDM '87: VDM - A formal method at work.* VDM-Europe Symposium 1987, Brussels, Belgium, 23-26 March 1987 (Berlin, Germany: Springer-Verlag 1987), pp. 237-59, 1987.

Minsky89 — Minsky N.H. & Rozenshtein D., 'Controllable Delegation: 'An Exercise in Law-Governed Systems'. In *OOPSLA '89. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, New Orleans, Louisiana, USA, 1-6 Oct. 1989 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **24**(10), pp. 371-380, Oct. 1989.

Miranda87 — Miranda E., 'BrouHaHa - A Portable Smalltalk Interpreter'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, Orlando, Florida, USA, 4-8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 354-365, Dec. 1987.

Moon86 — Moon D.A., 'Object-oriented programming with Flavors'. In *OOPSLA '86. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, Portland, Oregon, USA, 29 Sept. - 2 Oct. 1986, (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **21**(11), pp. 1-8, Nov. 1986.

Morrison87 — Morrison R., Brown A.L., Carrick R., Connor R.C.H., Dearle A. & Atkinson M.P., 'Polymorphism, persistence and software re-use in a strongly typed object-oriented environment', *Software Engineering Journal (GB)*, **2**(6), pp. 199-204, Nov. 1987.

Morrison87a — Morrison R., Barter C.J., Brown A.L., Carrick R., Connor R.C.H., Dearle A., Hurst A.J. & Livesey M.J., 'Polymorphic Persistent Processes'. Persistent Programming research report 39, University of St. Andrews, pp. 17, June 1987.

Morrison88 — Morrison R., Atkinson M.P., Brown A.L. & Dearle A., 'Bindings in persistent programming languages', *ACM SIGPLAN Notices (USA)*, **23**(4), pp. 27-34, April 1988.

Moss87 — Moss J.E.B., 'Managing Stack Frames in Smalltalk', In *SIGPLAN '87 Symposium on Interpreters and Interpretive techniques*, proceedings. *ACM SIGPLAN Notices (USA)*, **22**(7) , pp. 229-240, July 1987.

Naur76 — Naur P., Randell B., & Buxton J.N., (Eds), *Software Engineering Concepts and Techniques.* Petrocelli/Charter, New York 1976.

Nelson80 — Nelson T., 'Symposium On Actor Languages', *Creative Computing (USA)*, pp. 61-70, Oct. 1980.

Nygaard86 — Nygaard K., 'Basic Concepts in Object Oriented Programming', *ACM SIGPLAN Notices (USA)*, **21**(10), pp. 128-132, Oct. 1986.

Partsch83 — Partsch H. & Stenbruggen R., 'Program transformation systems', *ACM Computing Surveys (USA)*, **15**(3), pp. 199-236, Sept. 1983.

Pascoe86 — Pascoe G.A., 'Elements of object-oriented programming', *BYTE (USA)*, **11**(8), pp. 139-44, Aug. 1986.

Penney87 — Penney D.J. & Stein J., 'Class Modification in the GemStone Object-Oriented DBMS'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, Orlando, Florida, USA, 4-8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 111-117, Dec. 1987.

Poo91 — Poo C.-C.D., 'Representing Business Policies in the Jackson System Development Method', *The Computer Journal (GB)*, **34**(2), pp. 123-131, Feb. 1991.

Potts85 — Potts C., Bartlett A., Cherrie B. & MacLean R., 'Discrete Event Simulation as a means of Validating JSD design Specifications'. In *8th International Conference on Software Engineering.* Aug. 28-30, Imperial College London. IEEE Computer Society Press, pp. 119-125, 1985.

Pratt84 — Pratt T.W., *Programming Languages: Design and Implementation.* Prentice Hall, 2nd Edition, 1984.

Pun89 — Pun W.W.Y. & Winder R.L., 'A Design Method for Object-Oriented Programming'. In *ECOOP '89, Proceeding of the third European Conference on Object Oriented Programming*, (Ed. S. Cook), Nottingham, UK, 10-14 July. Cambridge University Press, pp. 225-240, 1989.

Put88 — Put F., *Introducing Dynamic and Temporal Aspects in a Conceptual (Database) Schema.* PhD Thesis, Katholieke Universiteit, Leuven, 1988.

Ratcliff87 — Ratcliff B., *Software engineering: principles and methods.* Blackwell Scientific Publications, 1987.

Ratcliff89 — Ratcliff B., 'A generalised inversion capability for the PRESTIGE workbench: Some basic issues'. Project PRESTIGE, Dept. Computer Science, Aston University, UK. (Document EXWP//3(1.0)300689), 1989.

Renold88 — Renold A., 'Jackson System Development for Real Time Systems', *Scientia Electrica (Switzerland)*, **34**(2), pp. 3-43, 1988.

Rentsch82 — Rentsch T., 'Object oriented programming', *ACM SIGPLAN Notices (USA)*, **17**(9), pp. 51-7, Sept. 1982.

Reynolds70 — Reynolds J.C., 'GEDANKEN: a simple typeless language based on the principles of completeness and the reference concept', *Communications of the ACM (USA)*, **17**(9), pp. 51-57 1982.

Rochat86 — Rochat R., 'In Search of Good Smalltalk Programming Style'. Technical Report no. CR-86-19, Computer Research Laboratory, Tektronix Laboratories, 17pp, September 1986.

Roper87 — Roper M. & Smith P., 'A structural testing method for JSP designed programs', *Software-Practice and Experience (GB)*, **17**(2), pp. 135-57, Feb. 1987.

Rowe81 — Rowe L.A., 'Data abstraction from a programming language viewpoint'. In *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling*, Pingree Park, CO, USA, 23-26 June 1980. *ACM SIGART News Letter (USA)*, no. 74, pp. 29-35, Jan. 1981.

Sakkinen89 — Sakkinen M., 'Disciplined Inheritance'. In *ECOOP '89, Proceeding of the third European Conference on Object-Oriented Programming*, (Ed. S. Cook), Nottingham, UK, 10-14 July. Cambridge University Press, pp. 39-56, 1989.

Sandberg86 — Sandberg D., 'An alternative to subclassing'. In *OOPSLA '86. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Portland, Oregon, USA, 29 Sept. - 2 Oct. 1986 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **21**(11), pp. 424-8, Nov. 1986.

Sanden89 — Sanden B., 'An Entity-Life Modeling Approach to the Design of Concurrent Software', *Communications of the ACM (USA)*, **32**(3), pp. 330-343, March 1989.

Schmucker86 — Schmucker K.J., 'MacApp: An Application Framework', *BYTE (USA)* **11**(8), pp. 191-193, Aug. 1986.

Schoch79 — Schoch J.F., 'An overview of the programming language Smalltalk-72', *ACM SIGPLAN Notices (USA)*, **14**(9), pp. 64-73, Sept. 1979.

Shlaer89 — Shlaer S. & Mellor S.J., 'An Object-Oriented Approach to Domain Analysis', *ACM SIGSOFT Software Engineering Notes (USA)*, July 1989.

Shankar80 — Shankar K.S., 'Data structures, types, and abstractions', *IEEE Computer (USA)*, **13**(4), pp. 67-77, April 1980.

Shaw80 — Shaw M., 'The Impact of Abstraction Concerns on Modern Programming Languages', *Proceedings of the IEEE (USA)*, **68**(9), pp. 1119-1130, Sept. 1980.

Shilling89 — Shilling J.J. & Sweeney P.F., 'Three Steps to Views: Extending the Object-Oriented Paradigm'. In *OOPSLA '89. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, New Orleans, Louisiana, USA, 1-6 Oct. 1989 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **24**(10), pp. 353-361, Oct. 1989.

Smoliar82 — Smoliar S.W., 'Approaches to Executable Specifications', *ACM SIGSOFT Software Engineering Notes (USA)*, 7(5), pp. 155-159, Dec. 1982.

Snyder86 — Snyder A., 'CommonObjects: An Overview'. In Special issue of the ACM SIGPLAN notices on the Object-Oriented Programming Workshop held at IBM Yorktown Heights, June 9-13, 1986. *ACM SIGPLAN Notices (USA)*, **21**(10), pp. 19-28, Oct. 1986.

Snyder86a — Snyder A., 'Encapsulation and inheritance in object-oriented programming languages'. In *OOPSLA '86. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Portland, Oregon, USA, 29 Sept. - 2, Oct. 1986, (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **21**(11), pp. 38-45, Nov. 1986.

Sommerville89 — Sommerville I., *Software Engineering*. Third Edition. Addison-Wesley, Wokingham, 1989.

Stefik86 — Stefik M. & Bobrow D.G., 'Object-Oriented Programming: Themes and Variations', *AI Magazine (USA)*, **6**(4), pp. 40-62, 1986.

Stein87 — Stein L.A., 'Delegation Is Inheritance'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications.* Conference Proceedings, Orlando, Florida, USA, 4 - 8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 138-146, Dec. 1987.

Stirling85 — Stirling C. 'Follow Set Error Recovery', *Software-Practice and Experience (GB)*, **15**(3), pp. 239-257, March 1985.

Storer88 — Storer R., 'Data-driven software design using inversion', *Information and Software Technology (GB)*, **30**(2), pp. 99-107, March 1988.

Strachey67 — Strachey C., *Fundamental Concepts of Programming Languages.* Oxford University Programming Research Group 1967.

Straw89 — Straw A., Mellender F. & Riegel S., 'Object Management in a Persistent Smalltalk System', *Software-Practice and Experience (GB)*, **19**(8), pp. 719-737, 1989.

Strom86 — Strom R., 'A comparison of the object-oriented and process paradigms', In Special issue of the ACM SIGPLAN notices on the Object-Oriented Programming Workshop held at IBM Yorktown Heights, June 9-13, 1986. *ACM SIGPLAN Notices (USA)*, **21**(10), pp. 88-97, Oct. 1986.

Stroustrup86 — Stroustrup B., *The C++ Programming Language.* AT & T Bell Labs, Murray Hill, New Jersey. Addison-Wesley, March 1986.

Stroustrup88 — Stroustrup B., 'What is object-oriented programming?', *IEEE Software (USA)*, **5**(3), pp. 10-20, May 1988.

Sun89 — Sun Microsystems, Inc. *A RISC Tutorial*, 1989.

Sutcliffe88 — Sutcliffe A., *Jackson System Development,* Prentice Hall 1988.

Sutherland63 — Sutherland I., *Sketchpad: A Man-Machine Graphical Communication System.* PhD Thesis, MIT, 1963.

Taenzer89 — Taenzer D., Ganti M. & Podar S., 'Problems in Object-Oriented Software Reuse'. In *ECOOP '89, Proceeding of the third European Conference on Object-Oriented Programming*, (Ed. S. Cook), Nottingham, UK, 10-14 July. Cambridge University Press, pp. 25-38, 1989.

Tesler84 — Tesler L.G., 'Programming Languages', *Scientific America (USA)*, **251**(3), pp. 58-66, 66, Sept. 1984.

Thatte86 — Thatte S.M., 'Persistent Memory: A Storage Architecture for Object-Oriented Database Systems'. In *Proceedings of the 1986 International Workshop on Object-oriented Database Systems.* Washington, DC, USA. IEEE Computer Society Press, pp. 148-159, 1986.

Thomas89 — Thomas D., 'What's in an Object', *BYTE (USA)*, **14**(3), pp. 231-240, 1989.

Tsichritzis88 — Tsichritzis D.C. & Nierstrasz O.M., 'Fitting Round Objects into Square Databases'. In *ECOOP '88, Proceedings of the Second European Conference on Object-Oriented Programming* (Eds. S. Gjessing & K. Nygaard), Oslo, Norway, 15-17 Aug. *Lecture Notes in Computer Science*, no. 322, pp. 283-299, Springer-Verlag 1988.

Turner86 — Turner D., 'An Overview of Miranda', *ACM SIGPLAN Notices (USA)*, **21**(12), pp. 158-166, 1986.

Ungar84 — Ungar D., 'Generation Scavenging: A non-disruptive high performance storage reclamation algorithm'. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. pp. 157-167, April 1984.

Ungar87 — Ungar D. & Smith R.B., 'Self: The Power of Simplicity'. In *OOPSLA '87. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, Orlando, Florida, USA, 4-8 Oct. 1987 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)* **22**(12), pp. 227-242, Dec. 1987.

Wegner86 — Wegner P., 'Classification in Object-Oriented Systems'. In Special issue of the ACM SIGPLAN notices on the Object-Oriented Programming Workshop held at IBM Yorktown Heights, June 9-13, 1986. *ACM SIGPLAN Notices (USA)*, **21**(10), pp. 173-182, Oct. 1986.

Wegner89 — Wegner P. Learning the language', *BYTE (USA)*, **14**(3), pp. 245-50, 252-3, March 1989.

Wetherell79 — Wetherell C. & Shannon A., 'Tidy Drawings of Trees', *IEEE Transactions on Software Engineering (USA)*, **SE-5**(5), pp. 514-520, Sept. 1979.

Wilson84 — Wilson A.D., 'Programs to Process Trees, Representing Program Structures and Data Structures', *Software Practice and Experience (GB)*, **14**(9), pp. 807-816, Sept. 1984.

Wilson87 — Wilson R., 'Object-oriented languages reorient programming techniques', *Computer Design (USA)*, **26**(20), pp. 52-62, Nov. 1987.

Wirfs-Brock88 — Wirfs-Brock A. & Wilkerson B. 'An Overview of Modular Smalltalk'. In *OOPSLA '88. Object-Orientated Programming Systems, Languages and Applications*. Conference Proceedings, San Diego, California, USA, 25 Sept. - 30 Sept. 1988 (Ed. N. Meyrowitz), *ACM SIGPLAN Notices (USA)*, **23**(11), pp. 123-134, Nov. 1988.

Wirth85 — Wirth N., *Programming in Modula-2*, Springer-Verlag, 1985.

Wolczko88 — Wolczko M.I., *Semantics of Object-Oriented Languages*. PhD Thesis, The University of Manchester, 1988.

Wolczko88a — Wolczko M.I., 'Introducing MUST - The MUSHROOM Programming Language'. The MUSHROOM Group, Dept. of Computer Science, The University of Manchester. Oct. 1988.

Yokote87 Yokote Y. & Tokoro M. 'Concurrent Programming in ConcurrentSmalltalk'. In *Object-Oriented Concurrent Programming*, (Eds. A. Yonezawa & M. Tokoro), pp. 129-158, MIT Press 1987.

Yelland89 Yelland P.M., 'First Steps Towards Fully Abstract Semantics for Object-Oriented Languages'. In *ECOOP '89, Proceeding of the third European Conference on Object-Oriented Programming*, (Ed. S. Cook), Nottingham, UK, 10-14 July. Cambridge University Press, pp. 347-364, 1989.

Yonezawa87 Yonezawa A. & Tokoro M. (Eds), *Object-Oriented Concurrent Programming*. MIT Press 1987.

Zave82 Zave P., 'An Operational Approach to Requirements Specification for Embedded Systems', *IEEE Transactions on Software Engineering (USA)*, **SE-8**(3), pp. 250-269, May 1982.

Zave84 Zave P., 'The operational approach versus the conventional approach to software development', *Communications of the ACM (USA)*, **27**(2), pp. 104-118, Feb. 1984.

Zave84a Zave. P., 'An Overview of the PAISLey Project — 1984', *ACM SIGSOFT Software Engineering Notes (USA)*, **9**(4), pp. 12-19, July 1984.

Zave85 Zave P., 'A distributed alternative to finite-state-machine specifications', *ACM Transactions on Programming Languages and Systems (USA)*, **7**(1), pp. 10-36, 1985.

Appendices

Appendix A. Code supporting `ReEntrantObject`.

Object subclass: #ReEntrantObject
instanceVariableNames: 'continuation savedContext parameter'
classVariableNames: ''
poolDictionaries: ''
category: 'Control-Mechanisms'

I am an abstract superclass representing re-entrant objects. I am able to return from any method (except suspend and resume) by sending myself the message suspend. Normally when a method returns, the context it returns from is lost. I save that context before returning in my instance variable 'savedContext'. When a sender wishes to continue the suspended methods activity from the point where it last returned (suspended), the message resume is used. NB. In order for this class to function, two additional protocol need to be filed in. These are sender: and contextCopy in class Context.

instance variables:

savedContext	*<Context or nil>*	*a copy of the receivers current context for future resumption.*
continuation	*<Context or nil>*	*the context which the receiver will return too.*
parameter	*<Object>*	*an object to facilitate the passing of parameters between suspend and resume*

Colin Lewis 01:12:89

ReEntrantObject methodsFor: resumption

resume
"Resume the activity of the receiver from the point where it
was last suspended. First test to see if there is any context
to actually resume. If there is them make the saved
context's sender to that of this context's sender, and then
make this context's sender to the saved context. Finally, nil
out the saved context. NOTE do not override this method ! ! !"

savedContext isNil ifTnue: [↑self exitBlock].
thisContext sender: (savedContext sender: thisContext sender).
savedContext ← nil

resume: aParameter
"Resume the activity of the receiver from the point where it
was last suspended. This implementation allows a parameter
to be passed to the receiver on resumption. NOTE do not
override this method ! ! !"

parameter ← aParameter.
self resume

resumeWith
"Resume the activity of the receiver from the point where it
was last suspended. Make a copy of the receivers sender
so we can, if needed, return to it. NOTE do not override
this method ! ! !"

continuation ← thisContext sender deepCopy.
self resume

resumeWith: aParameter

"Resume the activity of the receiver from the point where it was last suspended. Make a copy of the receivers sender so we can, if needed, return to it. This implementation allows a parameter to be passed to the receiver on resumption. NOTE do not override this method ! ! !"

```
continuation ← thisContext sender deepCopy..
parameter ← aParameter.
self resume
```

resumeWith: aParameter returnTo: aContinuation

"Resume the activity of the receiver from the point where it was last suspended. Save the parameter 'aContinuation' so we can, if needed, return to it. This implementation allows a parameter to be passed to the receiver on resumption. NOTE do not override this method ! ! !"

```
continuation ← aContinuation deepCopy.
parameter ← aParameter.
self resume
```

ReEntrantObject methodsFor: suspension

suspend

"Suspend the activity of the receiver in such a way that the receiver can be resumed, from the point where it was last suspended. NOTE do not override this method ! ! !"

```
self save: thisContext
```

suspend: returnParameter

"Suspend the activity of the receiver in such a way that the receiver can be resumed, from the point where it was last suspended. This implementation allows a parameter to be returned to the sender on suspension. NOTE do not override this method ! ! !"

```
self save: thisContext.
↑returnParameter
```

suspendTo

"Suspend the activity of the receiver in such a way that the receiver can be resumed, from the point where it was last suspended. Return control not necessarily to my sender, but to the continuation context held in 'continuation'. NOTE do not override this method ! ! !"

```
savedContext ← thisContext sender shallowCopy sender: nil.
↑thisContext sender: continuation deepCopy
```

suspendTo: returnParameter

"Suspend the activity of the receiver in such a way that the receiver can be resumed, from the point where it was last suspended. Return control not necessarily to my sender, but to the continuation context held in 'continuation'. This implementation allows a parameter to be returned to the continuation on suspension. NOTE do not override this method ! ! !"

savedContext ← thisContext sender shallowCopy sender: nil.
thisContext sender: continuation deepCopy.
↑returnParameter

ReEntrantObject methodsFor: tagged computation

escapeWith: aParameter

"Some condition has arisen in which we want to continue execution from the context saved during the tag: message. The first statement in this method merely helps the garbage collector to remove to chain of contexts by breaking their connections with each other. NB. Start releasing contexts from my sender, not here since this context would not be able to return the parameter at the end. Next make this context's sender to that of the saved context in continuation. Finally, return the passed parameter, aParameter."

thisContext sender releaseTo: continuation.
thisContext sender: continuation.
↑aParameter

tag: aComputation

"Tag the current contexts sender, so in the event of encountering an escapeTo: message we can continue execution from this context's sender."

continuation ← thisContext sender.
↑aComputation value

ReEntrantObject methodsFor: accessing

continuation: aContinuation

continuation ← aContinuation

parameter: aParameter

parameter ← aParameter

savedContext: aContext

savedContext ← aContext

ReEntrantObject methodsFor: private

exitBlock

```
self error: 'Can not resume'
```

save: aContext

```
"Suspend the activity of the receiver in such a way that the
receiver can be resumed, from the point where it was last
suspended. Do this by saving the context aContext in
savedContext. Break the chain of contexts which aContext
has, since on resumption, the copied context will have a new
sender assigned to it. (NB. it also reduces the size of the
object to be saved). Next make aContext's sender to that
of its sender's sender."

savedContext ← aContext sender shallowCopy sender: nil.
aContext sender: aContext sender sender
```

ContextPart methodsFor: ctrl mechanisms support

coroutineWith: aBlock

```
"Start the first coroutine off, and pass the second one as a parameter. Note
that the second coroutine block at present does not have any value for its
sender, so make it the context in which it was defined."

↑self value: (aBlock sender: aBlock home)
```

deepCopy

```
sender isNil
    ifTrue: [↑self shallowCopy]
    ifFalse: [↑self shallowCopy sender: self sender deepCopy]
```

sender: aContext

```
"Make the receivers sender aContext."

sender ← aContext
```

transfer

```
"Transfer control to my other coroutine. Although the implementation is very
small, its mechanics are quite subtle. First temporarily store the active
contexts sender (which will be a block, the other coroutine). Next, make the
value of the active contexts sender to that of the receiver. This can be done
because the receiver is a block context ([]). Finally, return as a value the
temporarily saved coroutine. When I return, I will return to the context
represented by the receiver of the transfer message, a block context. Since
the format of coroutining two blocks (see coroutine example) has a single block
parameter, returning temp here will become the coroutines parameter, which it
can subsequently transfer control to. Note that we do not use the value message
to evaluate the other coroutine."

| temp |
temp ← thisContext sender.
thisContext sender: self.
↑temp
```

Appendix B. Code for Class `ReEntrantExamples`.

ReEntrantObject subclass: #ReEntrantExamples
instanceVariableNames: "
classVariableNames: "
poolDictionaries: "
category: 'Control-Mechanisms'

I am a subclass of ReEntrantObject and so I am able to have any of my methods suspended and then resumed. I simply contain protocol which gives examples of the suspend and resume mechanisms in my abstract class.

ReEntrantExamples methodsFor: example-coroutine

firstCoroutine

```
Transcript show: 'In firstCoroutine for the first time'; cr.
self secondCoroutine.
Transcript Show: 'In firstCoroutine for the second tirne'; cr.
self resume.
Transcript show: 'In firstCoroutine for the third time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the forth time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the fifth time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the sixth time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the seventh time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the eighth time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the ninth time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the tenth time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the eleventh time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the twelfth time'; cr.
self resume.
Transcript show: 'In firstCoroutine for the last time'; cr
```

secondCoroutine

```
Transcript show: 'In secondCoroutine for the first time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the second time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the third time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the forth time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the fifth time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the sixth time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the seventh time'; cr.
self suspend.
```

```
Transcript show: 'In secondCoroutine for the eighth time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the ninth time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the tenth time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the eleventh time'; cr.
self suspend.
Transcript show: 'In secondCoroutine for the last time'; cr.
↑nil
```

ReEntrantExamples methodsFor: example-loop

loopTest

```
| count |
Transcript show: 'I am in the test'; cr.
count ← 0.
[true]
    whileTnue:
        [Transcript show: 'count value is :- ', count printString; cr.
        count ← count + 1.
        self suspend]
```

ReEntrantExamples methodsFor: example-continuations

powerFact: aStream

```
"Calculate the factorial of each (integer) element in aStream,
multiply the result by the original number. Do this for all the
elements in the stream and multiply all the results together.
NB. If a zero is encountered during the stream read, then
exit powerFact: immediately without unwinding the stack of
contexts which has been created during the successive
recursive calls of powerFact:. Only perform the actual
calculations when the entire stream has been read in while
the stack is unwinding."

| value |
aStream atEnd ifFalse: [↑ (value ← aStream next) = 0
    ifTrue: [self escapeWith: value]
    ifFalse: [(self powerFact: aStream)
        * value factorial * value]].
↑1
```

powerFactWithoutZero

```
"Calculate the powerFact function on the created stream. Tag this current
context so it can be returned to if a zero is encountered in the stream."

| aStream |
aStream ← ReadStream on: #(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
            25 26 27 28 29 30).
Transcript show: (self tag: [self powerFact: aStream]) printString; cr
```

powerFactWithZero

"Calculate the powerFact function on the created stream. Tag this current context so it can be returned to if a zero is encountered in the stream."

```
| aStream |
aStream ← ReadStream on: #(1 2 3 4 5 6 7 8 91011 12 13 14 15 16 17 18 19 20 21 22 23 24 25
        26 27 28 29 0 30).
Transcript show: (self tag: [self powerFact: aStream]) printString; cr
```

ReEntrantExamples class instanceVariableNames: "

ReEntrantExamples class methodsFor: examples

example1

"ReEntrantExamples example1."
"This is a simple test of the coroutining mechanism. Follow the output in the System Transcript."

```
| rObject |
rObject ← self new.
rObject firstCoroutine
```

example2: maxValue

"ReEntrantExamples example2: 50."
"This is a simple example of being able to suspend the execution of a method (loopTest) and resuming it from the point last suspended."

```
| rObject cnt |
cnt ← 0.
rObject ← self new.
rObject loopTest.
[cnt < maxValue]
    whileTrue:
        [rObject resume.
        cnt ← cnt+1]
```

example3a

"ReEntrantExamples example3a."
"This example shows the behaviour and use of continuations. The powerFact algorithm does not do any multiplications until all values have been read in off the stream. If at any time during the recursive call of powerFact a zero is encountered in the stream, powerFact simply returns to the outermost context (continuation) which has been tagged."

```
↑self new powerFactWithoutZero
```

example3b

"ReEntrantExamples example3b."
"This example shows the behaviour and use of continuations. The powerFact algorithm does not do any multiplications until all values have been read in off the stream. If at any time during the recursive call of powerFact a zero is*
encountered in the stream, powerFact simply returns to the outermost context (continuation) which has been tagged."

```
↑self new powerFactWithZero
```

Appendix C. Trace details of the Book Process.

Function	Result of Function	Rules used (from [3,2])
FIRST(Book)	= FIRST(Acquire)	- rule (b)(i)
FIRST(Acquire)	= {*Acquire*}	- rule (a)
FOLLOW(Acquire)	= FIRST(Classify)	- rule (b)(iii)
FIRST(Classify)	= {*Classify*}	- rule (a)
FOLLOW(Classify)	= FIRST(LoanPart)	- rule (b)(iii)
FIRST(LoanPart)	= FIRST(Loan) or FOLLOW(LoanPart)	- rule (d)(i)
FIRST(Loan)	= FIRST(Lend)	- rule (b)(i)
FIRST(Lend)	= {*Lend*}	- rule (a)
FOLLOW(LoanPart)	= FIRST(EndPart)	- rule (b)(iii)
FIRST(EndPart)	= FIRST(Sell) or FIRST(Dispose)	- rule (c)(i)
FIRST(Sell)	= {*Sell*}	- rule (a)
FIRST(Dispose)	= {Dispose}	- rule (a)
FOLLOW(Lend)	= FIRST(OutOnLoan)	- rule (b)(iii)
FIRST(OutOnLoan)	= FIRST(Renew) or FOLLOW(OutOnLoan)	- rule (d)(i)
FIRST(Renew)	= {*Renew*}	- rule (a)
FOLLOW(OutOnLoan)	= FIRST(Return)	- rule (b)(iii)
FIRST(Return)	= {*Return*}	- rule (a)
FOLLOW(Renew)	= FIRST(Renew) or FOLLOW(OutOnLoan)	- rule (d)(ii)
FIRST(Renew)	= {*Renew*}	- rule (a)
FOLLOW(OutOnLoan)	= FIRST(Return)	- rule (b)(iii
FIRST(Return)	= {*Return*}	- rule (a)
FOLLOW(Return)	= FOLLOW(Loan)	- rule (b)(ii)
FOLLOW(Loan)	= FIRST(Loan) or FOLLOW(LoanPart)	- rule (d)(ii)
FIRST(Loan)	= FIRST(Lend)	- rule (b)(i)
FIRST(Lend)	= {*Lend*}	- rule (a)
FOLLOW(LoanPart)	= FIRST(EndPart)	- rule (b)(iii)
FIRST(EndPart)	= FIRST(Sell) or FIRST(Dispose)	- rule (c)(i)
FIRST(Sell)	= {*Sell*}	- rule (a)
FIRST(Dispose)	= {Dispose}	- rule (a)
FOLLOW(Sell)	= FOLLOW(EndPart)	- rule (c)(ii)
FOLLOW(EndPart)	= FOLLOW(Book)	- rule (b)(ii)
FOLLOW(Book)	= Ø	- rule (f)
FOLLOW(Dispose)	= FOLLOW(EndPart)	- rule (c)(ii)
FOLLOW(EndPart)	= FOLLOW(Book)	- rule (b)(ii)
FOLLOW(Book)	= Ø	- rule (a)

The followmap of the Book process is:-

FIRST(Book)	= {Acquire}
FOLLOW(Acquire)	= {Classify}
FOLLOW(Classify)	= {Lend, Sell, Dispose}
FOLLOW(Lend)	= {Renew, Return}
FOLLOW(Renew)	= {Renew, Return}
FOLLOW(Return)	= {Lend, Sell, Dispose}
FOLLOW(Sell)	= Ø
FOLLOW(Dispose)	= Ø

Guards for the Book process

g(Book, Acquire) = true
g(Acquire, Classify) = true
g(Classify, Lend) = $Loan_c$
g(Classify, Sell) = not $Loan_c$ and $Sell_c$
g(Classify, Dispose) = not $Loan_c$ and $Dispose_c$
g(Lend, Renew) = $Renew_c$
g(Lend, Return) = not $Renew_c$
g(Renew, Renew) = $Renew_c$
g(Renew, Return) = not $Renew_c$
g(Return, Lend) = $Loan_c$
g(Return, Sell) = not $Loan_c$ and $Sell_c$
g(Return, Dispose) = not $Loan_c$ and $Dispose_c$

Appendix D. Smalltalk-80 representation of the BookProcess.

```
JSD subclass: #BookProcess
    instanceVariableNames:  'dateAcquired title purchasePrice lastBorrower
                             lendCount lendDate dateOfLastRenewal inLib
                             totalTimeOnLoan author '
    classVarlableNames:     'DS1 DS2'
    poolDictionaries: ''
    category:               'JSD-Process'
```

BookProcess methodsFor: operations

acquire

"The library acquires the book"

```
key ← parameters at: 1.
dateAcquired ← parameters at: 2.
title ← parameters at: 3.
purchasePrice ← parameters at: 4.*
author ← parameters at: 5.
lendCount ← 0.
totalTimeOnLoan ← 0.
inLib ← false.
DS2 write: #acquire.
DS1 read
```

classify

"The book is classified and catalogued"

```
inLib ← true.
DS2 write: #classify.
DS1 read
```

dispose

"The book is disposed of. It is no longer on the library catalogue"

```
inLib ← false.
DS2 write: #dispose
```

lend

"Someone borrows a book (the library lends it)"

```
lendCount ← lendCount + 1.
lendDate ← Time totalSeconds.
inLib ← false.
DS2 write: #lend.
DS1 read
```

renew

"The borrower renews the loan on a book"

```
lendCount ← lendCount + 1.
dateOfLastRenewal ← Time dateAndTimeNow.
DS2 write: #renew.
DS1 read
```

return
"The borrower returns the book to the library"

inLib ← true.
totalTimeOnLoan ← totalTimeOnLoan + (Time totalSeconds - lendDate).
DS2 write: #return.
DS1 read

sell
"The book is sold"

inLib ← false.
DS2 write: #sell

BookProcess methodsFor: guards

acquireG
self state = #book ifTrue: [↑true]

classifyG
self state = #acquire ifTrue: [↑true]

disposeG
self state = #classify ifTrue: [↑self loanC not and: [self disposeC]].
self state = #return ifTrue: [↑self loanC not and: [self disposeC]]

lendG
self state = #classify ifTrue: [↑self loanC].
self state = #return ifTrue: [↑self loanC]

renewG
self state = #lend ifTrue: [↑self renewC].
self state = #renew ifTrue: [↑self renewC]

returnG
self state = #lend ifTrue: [↑self renewC not].
self state = #renew ifTrue: [↑self renewC not]

sellG
self state = #classify ifTrue: [↑self loanC not and: [self sellC]].
self state = #return ifTrue: [↑self loanC not and: [self sellC]]

BookProcess methodsFor: conditions

disposeC
↑message = #dispose

loanC
↑message = #lend

renewC
↑message = #renew

sellC
↑message = #sell

BookProcess methodsFor: private

author
    ↑author

author: anObject
    author ← anObject

dateAcquired
    ↑dateAcquired

dateAcquired: anObject
    dateAcquired ← anObject

dateOfLastRenewal
    ↑dateOfLastRenewal

dateOfLastRenewal: anObject
    dateOfLastRenewal ← anObject

inLib
    ↑inLib

inLib: anObject
    inLib ← anObject

lastBorrower
    ↑lastBorrower

lastBorrower: anObject
    lastBorrower ← anObject

lendCount
    ↑lendCount

lendCount: anObject
    lendCount ← anObject

lendDate
    ↑lendDate

lendDate: anObject
    lendDate ← anObject

purchasePrice
    ↑purchasePrice

purchasePrice: anObject
    purchasePrice ← anObject

title
    ↑title

title: anObject
    title ← anObject

totalTimeOnLoan
↑totalTimeOnLoan

totalTimeOnLoan: anObject
totalTimeOnLoan ← anObject

BookProcess class instanceVariableNames: "

BookProcess class methodsFor: inst var comments

authorComment
↑'This is the author of the book' asText

dateAcquiredComment
↑'The date at which the book was acquired by the library' asText

dateOfLastRenewalComment
↑'The date the book was last renewed for borrowing' asText

InLibComment
↑'Boolean value representing whether or not the book is currently in the library' asText

lastBorrowerComment
↑'The unique identifier accessing the last borrower of this book (ie will reference a memberid)' asText

lendCountComment
↑'The number of times this book has been lent to members of the library since the book was acquired' asText

lendDateComment
↑The date the book was last lent out' asText

purchasePriceComment
↑'The price of the book in british sterling' asText

titleComment
↑'The title of the book' asText

totalTimeOnLoanComment
↑'Value indicating the total amount of time this book has been has been out on loan since it was acquired' asText

BookProcess class methodsFor: private

DS1
↑DS1

DS1: anObject
DS1 ← anObject

DS2
↑DS2

DS2: anObject
DS2 ← anObject

keyField
↑#isbn