1. Jackson System Development

1.1. An Overview

1.1.1. Extension of JSP

Jackson System Development (JSD) was developed as a superset, or an enlargement of Jackson Structured Programming (JSP). Whereas JSP is a method essentially for designing and developing single programs, JSD takes the process a stage further by allowing the designer to build complete computer systems of communicating sequential programs. JSD is significantly oriented towards effective software production and hence its critics say it does not sufficiently address feasibility and cost justification.

1.1.2. Theoretical Foundations

JSD is founded on the principle that the development of a system should consider the real world subject matter about which the system is to compute and produce information and should construct a formal model of that subject matter, before turning attention to the functional requirements of the system. The most relevant aspect of real world subject matter is when something happens to it and it is this aspect of happening (or the occurrence of an action) which the system developer should be most interested in understanding when starting a development project.

1.1.3. Real World Modelling

JSD is ideal for developing systems in whose real world subject matter it is important to recognise the occurrence of actions within a time dimension. Control systems, embedded systems, data processing and information systems all fall into this class. The central aspect of JSD is the modelling of the real world and the use of the resultant model as the basis for later development stages.

1.1.4. Limitations

JSD requires that the problem have a time dimension. This implies that certain systems (such as a census) cannot be modelled using JSD.

1.1.5. JSD and Maintainability

Jackson claims that it is easier to build maintainable systems using JSD because the initial model of the real world is a stable basis for development, being less likely to change than the functions subsequently added to it. Should additional functionality be required then a model created using JSD is easier to add to than a model based on hierarchical decomposition of functional requirements. This is because the initial JSD model does not consider the functional requirements.

1.1.6. JSD Stages

JSD identifies three phases which broadly relate to the system specification, design and construction stages of the software lifecycle. These are:
1. The Modelling Phase;
2. The Network Phase;
3. The Implementation Phase.

The first of these, the modelling stage, comprises the main thrust of the method and is the most widely used aspect of the method in industry.

1.2. Modelling Phase

In the modelling phase, the developer identifies those actions of interest in the application domain. An action is simply an instantaneous happening, such as the arrival of a customer order or the point at which somebody’s account is opened. For each action, a list of data attributes can be defined. For example, the purchase of a book would have the attributes, book title, author, price, etc.

The next step within the phase is to take each action and try to put it in time order, under a given entity name, so that each action relates to an entity.

1.2.1. The entity in JSD

An entity is something in the real world that performs actions on the system or suffers actions from the system.

1.2.2. The Steps

1. From analysis notes, interviews and documentation draw up a list of actions which may be found in phrases, verbs and nouns.;
2. Describe actions with one or two sentences of English and list actions attributes;
3. Allocate actions to entities;
4. Create preliminary action sequence in entity life histories;
5. Review entities to check: entity boundaries are correct, and no separate lives exist (marsupial entities);
6. Describe entity life histories as a model process with a process structure diagram (a process structure diagram follows the same conventions as for JSP);
7. Define entity attributes, the type and updating actions.

1.2.3. Documentary Output

1. Candidate list of actions and entities and reasons for any rejected items;
2. Final list of actions and entities;
3. Action descriptions and associated attributes;
4. An entity/action cross reference;
5. At least one process structure diagram for each entity. Figure 1-1 is an example of a process structure diagram or model process for the entity CUSTOMER with each elementary component representing an action the customer performs;
6. The Structure Text (schematic logic).
1.3. Network Phase

Once the entity structures have been identified, the network phase is concerned with two aspects of the real world model. Firstly, the aim is to take each model process and think about the relationship between processes. This produces a network model which shows the relationships between model processes. These relationships are identified in terms of the data examined or exchanged between model processes.

Secondly, this is followed by the analysis of the output subsystem which provides the output functionality required from the modelled system and the input subsystem which handles the user interface and input validation.

1.3.1. The Steps:

- The initial model phase;
- The elaboration phase.

1.3.2. Initial Model Phase

- Using action attributes, list the model process inputs;
- Examine model process actions for outputs;
- Identify common events;
- Determine which type of connection to use, depending on whether a short-term or long-term view is required. A short-term view is a snapshot interested in the current state of an attribute. A long-term view is an action history interested in all changes of state of the attribute. Use state vector connections for the former and data stream connections for the latter;
- Construct the initial model, showing model processes with their inputs and outputs be they state vector or data stream connections.

Documentary Output

- A revision of the documentation from the modelling stage;
- A system specification diagram (SSD) showing the connections between the
1.3.3. Elaboration Phase:

The elaboration phase adds the functionality required from the system to the SSD. Once we have designed the majority of our system we then add the new functionality to cover the user requirements. This new functionality is a set of new processes or amendments to existing model processes to incorporate the functionality. These new processes are known as functions or functional processes. There are four types of functional process: embedded functions, output functions, interactive functions and filter functions.

Embedded Functions: These alter an existing model process. If no structural change is required then they may be embedded into an existing model process. They usually consist of the addition of variables, elementary operations and so forth to an existing model process. Small subtrees may be added provided the structural integrity of the model process is maintained. Any number of new variables and elementary operations may be added but nothing must be done to change the time ordering of the model processes actions. They may be connected to the rest of the SSD by either data stream or state vector connection.

Output Functions: These are similar to embedded functions in that they are present to fulfil some other function or user requirement. They are modelled as separate processes. Consider the embedded function which, by the addition of some actions, sends data for printing or screen output. Obviously there needs to be a report formatting process. The output function receives the raw data from the embedded function and then formats it for output. Output processes must frequently inspect the state vector of their connected processes and are usually of a report writing or query screen nature. Since most state vectors are in implementation terms files many database query programs are in effect output functions. Since most output functions are usually of a query nature and therefore stimulated by user demand rather than at a given time, state vectors are the obvious form of connection. When the user demands to see something it is the current state that is required, not what happens to be next in the buffer of a data stream. However, data streams may be used if the query is of a mandatory monitor nature. Output functions usually have to inspect the state vectors of many processes and usually in a strict order, query functions tend to be complex.

Interactive Functions: This type of function is also an individual process, usually because of the number of tasks involved in the type of user requirement which leads to the need for the function. They are the most complex form of function. Interactive functions are so called because they interact with the system and/or user; they gather information from the user and/or other systems, process it and feed it back into the system. The connections to the rest of the system are usually data streams and usually involve many new and changed actions being embedded into existing processes. Data streams are used rather than state vectors because the processing of data is usually in some mandatory sequence.

Because of the complex nature of interactive functions they are necessarily large, forming complete sub-systems in their own right. Although they carry out the requirements of a complex interactive user function (either with the system or the user) they should be built as if they are systems i.e. we use most, if not all, of the stages of the development cycle, starting with the requirements capture stage.

Filter Functions: These are used to filter out invalid data from or within the system. Filter function literally filter input data to the processes connected to the outside world or other sources of potentially invalid data. Filter functions are always connected to other processes via data streams which enforce a required closely coupled relationship. Filter functions reduce the complexity if the model processes by modelling most of the bad data possibilities.

Timing Constraints: There are a number of ways that timing constraints can be added to the model. the analysis of the system will have identified those processes which need constraint. For example, we need to be able to schedule when a report is required; we may need to synchronise groups of processes and so on. If more than one data stream is inputting into a process and the order in which data arrives is not important then the data streams can be rough merged. How-
ever, data may be required more quickly from one data stream than another, in which case some control needs to be exerted.

The overall objective of adding time constraints to the system is to bring about synchronisation between processes and within processes. At the network level synchronisation effectively makes an otherwise concurrent system behave sequentially. It is worth remembering that individual processes are designed as essentially sequential whilst the network model is primarily concurrent. Therefore, the sequential flow of control is theoretically uninhibited, the only constraints being the speed of the processor (which is an implementation problem), and the availability of data external to the process.

Consider a process that is processing input data—eventually it must produce some output. When does it produce this output? At a given time? How much data must it process first? Which input data is vital, which is optional? We could go on. The most critical aspect of these varieties is time. Time is entered into the system like any other piece of data but always as a data stream—this is known as a time grain marker (TGM). TGMs are sent to a process as a separate specialised data stream and are read as input records. The process looks for their arrival before some action is allowed to occur. A frequent use of TGMs is the control of rough merged data streams. Rough merged data streams exhibit no favouritism to any of the constituent data streams, and if one is writing faster than the other then it will tend to get more attention. TGMs can control this state of affairs.

**Documentary Output:**

- A SSD showing the extra functional processes, their connections with the model processes and their connections with each other. Figure 1.3 is a sample SSD showing the process CUSTOMER connected to the process ACCOUNT by a data stream connection; and the process BALANCE REPORT connected to ACCOUNT by a state vector

![Customer AccountDS Balance Report SV](image)

**Figure 1.3: A Sample System Specification Diagram**

- Elaboration of the model process documentation (structure diagram with operations allocated and list of operations) to show extra local variables and calculations used for embedded output or to hold information for state vector inspection;
- The structure and text of the extra functional processes;
- A concise statement of access paths on state vectors defined by functional processes.
1.4. Implementation Phase

In this phase the SSD, which is viewed as a network of concurrently communicating processes, is transformed into a sequential design that may be implemented on one or more processors. This process is known as scheduling. This is followed by further detailed design and coding.

The steps of this stage are:

- Divide the SSD into scheduling groups according to system requirements of distributed processing, hardware, and user interface needs;
- Add a scheduler process for each scheduling group;
- Start with system inputs and invert input processes as subprocesses of the scheduler;
- Following data stream connections in the SSD, invert the nearest neighbours to input processes as the next subprocess level and continue until output is reached;
- Add buffers for fixed merges and show access paths to processes;
- Add state vector files and show access paths to processes;
- Convert process specifications to inverted form by changing writes and reads to calls and exit-programs;
- Assign elementary operations and conditions to program components;
- Add further details to elementary operations and conditions in pseudo code or as statements in the target language.

1.4.1. Documentary Output:

- A set of informal timing constraints on the implementation;
- Elaboration of process structure diagrams;
- A system implementation diagram (SID).
1.5. JSD Notation

- Customer
- Entity/Process
- SV
- State Vector
- DS
- Data Stream
- TGM
- Time Grain Marker
- Connection 1:1
- Connection 1: Many
- Clock
- Clock Process
- Rough Merge
- Fixed Merge

1.6. Exercises

1. Why might JSD be called a method involving composition rather than the more usual decomposition?
2. Why might it be better to expend a good deal of time in the modelling phase rather than attempt to establish system functions early on?
3. Why do we say that JSD is a superset of JSP, rather than a front end to it?
4. What do you understand by the notion of time ordered entities?
2. JSD Modelling — Underlying Ideas

2.1.1. The Scope of JSD

Jackson system Development (JSD) can be used to develop systems in whose real world subject matter it is important to recognise the occurrence of events within a time dimension. Control systems, embedded systems, data processing and information systems all fall within this category. A central aspect of JSD is the modelling of the real world and the use of the resultant model as the basis for later development stages.

2.1.2. Towards More Maintainable Systems

“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. The developed system can then be expected to allow easy enhancement for functions consistent with this view of reality” ¹

Jackson claims that it is easier to build maintainable systems using JSD because the model of the real world is a stable basis for development, being less likely to change that the functions built upon it. That is to say that customers and their actions rarely change, but the way in which these actions are performed changes more frequently. The system developer and the user can more easily agree on a set of system functions to be supported either in original development or later maintenance because the system embodies the model and the model defines a set of possible functions. By contrast, a system which is not based on an explicit model of reality means that new functions may be arbitrarily cheap or expensive to implement.

2.2. Modelling the Real World

2.2.1. Modelling Before Functions

Traditionally system development starts by considering the functional requirements of the proposed system; often each major system function is organised in a hierarchy of processes connected by data flows. The development of systems using JSD starts by building a model of the real world which provides the subject matter of the system. the model is a realisation of an abstract description of the real world a kind of simulation. The function of the system are added to the model and the systems can produce a display of the state of the model or the occurrences of certain events which can be used for information or control purposes.

Consider the following example:

A car hire firm has recently installed a terminal which is used to record the dates when cars are hired out and returned. When a car is taken out the registration number is keyed in with a prefix [ to indicate taken out and the date (in the form of number of

days since system commission date) is automatically added. When a car is re-
turned the registration number is keyed in with the prefix ] and the date is again
amatically added. The manager wants the following information from the sys-
tem: the number of separate hire contracts and the average number of days fro
which a car is on hire

The functions of such a simple management reporting system are (ignoring the possible argu-
ment that data acquisition is itself a function):
i. Compute the information;
ii. Output it in a simple form.

The computation consists of three parts:
i. Count the number of separate hire contracts (i.e. count the transactions beginning with [);
ii. Total the number of days when cars are hired;
iii. Compute the average.

This could be represented in structured English as:

```
BEGIN
    initialise number of contracts and total days = zero;
    get first transaction;
    WHILE not end of transactions DO
        IF car taken out (code = '[') THEN
            number of contracts := number of contracts + 1;
            total days := total days-taken out date;
        ELSE
            total days + brought in date;
            get next transaction;
        END
    END
    PRINT number of contracts;
    IF number of contracts not = zero THEN
        print average duration (total days / number of contracts);
END
```

Examine the above solution to see if it works. Note, it is rather simple minded in the sense that
the management information will only be produced when all transaction data are available and
all cars are returned.

i. Now specify the changes that would be required if the manager now wanted a further report
giving the longest period of hire.

ii. Now specify the changes that would be required if the manager wanted two separate reports
one for hire periods of less than a week and one for a week or longer.

iii. What changes would be required to allow the system to be run at any time (i.e. when there
are cars outstanding)?

These changes are difficult without a complete rewrite because the system reflects the computer
expert's interests in algorithms to compute efficiently total elapsed time and does not consider
the user's main concern—the time for individual contract hire periods. The real world of the
manager is about contracts and their duration; a model of this real world would have revealed this
and implied possible functions some of which would have been realised when the manager made
his information requests which were all about contract hire periods.

Using JSD we would model a contract and ask questions about what happens to it. We then ob-
tain events, apply time ordering and use the results as a model process upon which to build pos-
sible functions.
2.2.2. Simulation of Objects in the Real World

Many software systems can be usefully regarded as simulation of their real world subject matter. Embodied in such a system is a discrete simulation of the behaviour of objects in the real world. The information and control objects of the system are in fact derived from the internal behaviour and state of the simulation but are interpreted as having validity in the real world. Thus we can inspect the state of the model to gain information.

Consider the traditional payroll system. The real world for this system is employees of the organisation, the work they do, the holidays they take, their arrival at the place of work and their departure from it, their promotions, their increasing age as the years go by, and so on. The essential outputs of the system are payslips and pay cheques, management reports about total pay, amount of overtime worked, perhaps also about productivity, and taxation and returns to such bodies as the Inland Revenue.

2.2.3. The Validity of the Outputs

The validity of the outputs is only to the extent that the system is running a valid simulation of the real world. If there are no simulated apprentices in the system, then there should not be any output in respect of apprentices. Conversely, when a real employee goes on holiday then a simulated employee also goes on simulated holiday in the computer system. In this case we are entitled to count the occurrences of the simulated holidays and draw inferences about real world holidays. Note here the similarity in concept to that of the object oriented paradigm where real world objects have their lives modelled and traced by system objects.

2.3. The Need for a High Quality Model

2.3.1. Accuracy for the Intended Purpose

The quality and value of an information or control system will depend very heavily on the accuracy of the simulation for the intended purposes. If we simulate the hours spent in the work place by events such as *Jones has spent 6 hours at work today*, we will be able to draw correct inferences about the total hours spent provided we wait until the end of the day. However, we will not be able to draw correct inferences during the working day about who is or who is not presently at work unless we simulate the arrivals and departures of the employees as they happen. Then, of course, we could inspect the state of the *Jones* process and ascertain whether he is at work, and if so how long he has been there.

2.3.2. The Model Must be Understood by Users

The model must also be understood by the people who are to use its outputs. A disparity between the understanding of the users of the systems and the system builders will lead to a loss of value in the information produced. Hence, the simulation must reflect the real world as the users of the information system see and understand it. In JSD communication between the developer and the user is improved because the developer is seem to be seeking knowledge from the user's point of view. A description of what the user already knows preceded invention of functional or technical novelties i.e. clever ways of simulating events in the real world. However, we must be aware that different system users will have different perceptions of the system. These differences being caused both by their differing viewpoints and personal subjectivity these differing
views are known as *roles*.

### 2.3.3. Models of the Real World

It is of the up most importance in building software systems of the kind we are considering here to take the greatest care in describing the real world and the model or simulation that will be built on the basis of that description. JSD concentrates heavily on modelling in the same sense that an architectural model is a model of a real building, or a map is a model of a real terrain. This is not modelling the system itself, or modelling some existing manual or mechanised system that is to be replaced, or modelling the behaviour of the system’s eventual users, but modelling the real world, the subject matter about which the system computes, and in the context of which its outputs are to be interpreted.

### 2.3.4. Careful Modelling

The model of the real world must be made in the most careful way using the descriptive languages and concepts that are best suited to the subject matter because this model will furnish the basis of the system’s ability to serve its ultimate purpose, the production of useful and valid information and control outputs.

### 2.4. Data Modelling

#### 2.4.1. The Entity Relationship Model

One of the sets of languages and concepts often used for modelling the real world is that offered by some form of data modelling such as the entity relationship model. Here, in essence we describe the real world by describing the entities to be found there, their attributes and the relationships which exist among them.

For example, if we are concerned with a world in which we engage in construction projects using items obtained from various suppliers, we might define a relation SIP (Supplier Item Project):

<table>
<thead>
<tr>
<th>S1: SUPPLIER</th>
<th>I: ITEM</th>
<th>P: PROJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>S123456</td>
<td>I678</td>
<td>P543210</td>
</tr>
<tr>
<td>S123456</td>
<td>I789</td>
<td>P543210</td>
</tr>
<tr>
<td>S123456</td>
<td>I678</td>
<td>P379106</td>
</tr>
</tbody>
</table>

A tuple in the relation SIP such as (S123456, I678, P543210) means that supplier S123456 *is-a-supplier* of item I678 to project P543210.

What exactly does *is-a-supplier* mean? If a catastrophe occurs in project P, and we want to warn all of the project suppliers that we might be suing then for damages, should we warn every supplier who has a tuple in this relation in which the column has the value P? (Why might this not be valid?) If the design of the project’s product is changed, and we want to request specification changes from suppliers of affected items, which suppliers must be told? To answer these and other similar questions we would naturally discuss the meaning of the relation in terms of events. We might say that a supplier S is *is-a-supplier* of item I for project P if S has been put on the approved list for I for project P and has not been removed from the list; or if S has delivered at least one batch of item I to project P; or if item I has been ordered from supplier S on at least one occasion for project P, even if it has not yet been delivered. From this it can be seen how crucial time-ordering is to interpretation.

#### 2.4.2. Real world events

The natural way to understand the meaning of the relation is to consider the real world events that affect it, and the natural way to explain the relation is in terms of those events. It follows
that in this example the occurrences of events are more basic than the existence of the relation, and that our modelling should start from the events. This will be true generally in systems whose real worlds exhibit the occurrence of events within a time-ordering: where there are events, where there is a time dimension, we need to start there.

2.5. Events in JSD

A JSD event is thought of as *atomic*: we do not decompose events into sub-events and we think of them as being instantaneous, occurring at a particular moment in time. This is, of course, an abstraction: our description of the real world will, necessarily, ignore some aspects and details of reality. This is comparable to the logical data structures used in JSP which ignore detail not of direct interest to the problem. To use a simple and well known illustration, the lending of a nook by a lending library may be described in a JSD model as an event; but the reality is more complex, because there is a whole microcosmic history from the moment the borrower takes the cook from the shelf and begins to browse in order to decide whether to borrow it up to the time the book is placed in the library's checkout desk and then, eventually, handed to the borrower to take home. We abstract from these complexities at a single point in time.

2.5.1. Event Attributes

The attributes of an event are, in essence, the answers to the sensible questions we might reasonably ask about the event occurrence. Suppose we are told a lend event has occurred. We might reasonably ask: What was lent?, and To whom was it lent?, and When did it happen? so the lend event may be considered to have attributes Borrower, Book (if we assume our library does not lend anything else other than books), and Date. By identifying events and their attributes in this way we are beginning to construct a description of the real world of the library system.

2.5.2. Event Types or Classes

Eventually we will have identified a set of event types or classes (the term class will apply to entities and events which have multiple occurrences), and the attributes associated with each. For each event class we will specify a class name, and names for the attributes; and we will give an informal narrative explanation of the event in which the meanings of the attributes are explained.

We might have in our library system:

**ACQUIRE (Book_Id, Date, Price, Author, Title, ISBN)**

The library acquires one book whose identifier is its book_id. The date of acquisition is the date on which the book arrives in the library stock reception room; Price is the price paid to the supplier, net of any postage, packing, or delivery charges; author, title and ISBN have their usual meanings.

**CATALOGUE (Book_Id, Date, Class)**

The librarian enters one book in the library catalogue, date is the date of entry into the catalogue; class is the Dewey Decimal number plus the first 3 characters of the author's name; Book_Id is the identifier of the book entered in the catalogue.

**LEND (Book_Id, Member_Id, Resvn_Id, Date)**

A book, whose identifier is Book_Id, is lent to the member whose membership
number us Member_Id; date is the date of removal of the book from the library's premises; Resvn_Id is the identifier of the relevant reservation if the book is borrowed against a previous reservation.

There may be many other definitions of event classes. Naturally, it will be appropriate to hold these definitions in some kind of development database (such as Speedbuilder q.v).

2.5.3. The Importance of Events

Three points need to be made about the importance of events in JSD:

1. Our initial description of the real world is in terms of what happens. We begin by describing the classes of events of interest. In the kind of real world we are talking about and most real worlds for information and control systems are of this kind, this is the most natural and direct description to give;

2. By listing event classes in this way, we are beginning to define the scope of the eventual system: the system will, in principle, be able to produce any outputs that can be deduced from knowledge of the occurrences of the events of the defined kinds and no other outputs;

3. We give an informal narrative description of each event class to provide the necessary link between the formal systems specification we are making and the real world itself as understood and experienced by our users and customers. Such a link between formal and informal is always needed unless the real world is itself a formal object or system.

2.6. Objects, Entities and Event Orderings

2.6.1. Choosing Entity (or Object) Classes

A description of the real world in terms of events only is likely to be very incomplete. Most often we will also want to describe the objects or entities that persist over time and that take part in the events: in the library we will want to describe at least books and members and perhaps other objects or entities too. In many cases we can recognise quite easily some or even all of the entity classes that we need to describe; but we will often find ourselves in doubt whether to include some particular candidate in the list of entity or object classes. This is arguably the hardest part of JSD. For that reason, and also because it is an illuminating exercise, we will approach the problem of choosing entity or object classes strictly on the basis of the set of event classes we have already constructed.

2.6.2. Event History

If we consider the set of all possible traces of events in the real world (where a trace is an ordered history of particular event occurrences), we may ask whether any regularities of ordering can be observed to be true of all traces. In particular we may ask this question about sets of events that share a particular attribute value: for example, about the set of events that all have the Book_Id value B4567, or the set of events that all have the Member_Id value M123. When considering such a set we will often be led to observe quite definite regularities of ordering: for example, that book B4567 must necessarily be ACQUIRED before it is LENT, and that LEND and RETURN events must alternate, and that various other ordering regularities also hold. We can express such regularities as time-orderings of events, for which a suitable notation is available in the textural and diagrammatic forms of regular expressions used in JSP program design.

2.6.3. Entity Diagrams

Figure 2.1 shows a book in our imaginary library: it expresses the time-orderings that are true in the particular real world are considering here, and we are assuming that our user (who knows all about the world of the library), has given us the information on which the diagram is based and
has confirmed that we have used that information correctly. N.b., Observe that
the diagramming rules for JSP apply equally to JSD

![Entity Structure Diagram](image)

Figure 2.1: An entity structure diagram

2.6.4. Identifying Entities

The fact that this is a time-ordering in the real world is a direct indication that
books must be regarded as entities in our real world description. Something
about the real world constrains LEND and RETURN events for a given Book_Id
value to alternate: that something is, of course, the nature of the LEND and RE-
RETURN events and the physical nature of books. Books are entities that have
states and these states change when books take part in events. Consider, by
contrast, the absence of time-ordering for a particular value of the process at-
tribute. We will not find that ACQUIRE events at price $X$ must precede, or fol-
low, or alternate with, SELL events at price $X$: there is no constraint at all on
the orderings of events with the attribute price = $X$. We may confidently assert,
therefore, that in the world of our library, price is not an entity class, and that
we will not wish to speak of price objects or price entities. (Of course, we are
making sensible assumptions for purposes of explanation; there is no reason
why in some other real world, prices should not be entities or objects and have
significant time-ordering).

2.7. Entity Attributes

2.7.1. Entity States

Having identified, perhaps, books, members, and reservations as significant en-
tity classes in our tiny library world we can turn our attention to describing
their states. The state of an entity is recorded, essentially, in the values of its
entity record fields or, in object-oriented parlance, its instance variables. As a
first step towards defining the system’s information outputs we may ask our-
selves what information we want each class of entity to store in its entity state.
This information is essentially just a summary record of the entity’s event his-
tory; we record it because we do not want to record all event occurrences over
the whole life of the system, and we do not want to be forced to analyse a com-
plete event history to answer every request for information.

---

2 In computerised stock market trading a ‘stop-loss’ event is triggered by a price event.
2.7.2. Specifying Entity Attributes

In JSD we specify entity attributes and their meanings in terms of the events we have already specified in the context of the time-orderings we have already given for the entities. Suppose, for example, we wish to specify an attribute of book that we may call *loan-count*: it is simply a count of the number of times the book has been lent. We would specify *loan-count* by saying how its value is determined by the events in the life of a book: it is zero when the book is initially *ACQUIRED* and its increment by one on each *LEND* event. Or should it be incremented by one on each *RETURN* event instead? The decision is one that we and our user may take freely in the light of the expected use to which this information will be put; but however the decision is made we will have an unambiguous and clearly agreed specification of what *loan-count* means.

A slightly more subtle example of the same point arises with an attribute *in-the-library*. We want to be able to ask of any book *is it in the library?* and we need to be sure that we and the users of the system have the same understanding of what *in-the-library* means. We achieve this certainty by specifying the value of *in-the-library* at every point in the history of a book and so eliminate the possibility of misunderstanding altogether. An example of a possible misunderstanding might be: Is a book *in-the-library* when it is out on loan?

2.8. Exercises

1. Why is it claimed that JSD facilitates systems maintenance? If the claim is true, what is its significance?
2. How would you explain the statement a JSD model is a realisation of an abstract description of the real world?
3. Use a simple (?) payroll system to show how validity of the systems outputs is crucially affected by the quality of the model or simulation of the real world.
4. If the system's developer and the user have different understandings of the computer system's behaviour any information produced is suspect. Is this statement true? Does JSD do anything to help overcome this problem?
5. In what sense are JSD models models of the real world? With what type of models should they not be confused?
6. Explain why the entity relationship model may not be ideal in some cases.
7. What is a JSD event?
8. How might we identify event attributes?
9. How might we advantageously define events?
10. How important is it to start a requirements analysis by considering the events of the system?
11. What will a time-ordering of events allow us to produce?
12. How might a JSD entity be recognised?
13. What is an entity attribute? Give some examples.
14. How might we clarify the definition of an entity attribute?
3. The JSD Modelling Stage

3.1. Definitions

Let us first try to establish what Jackson means by Entity and Action, because they are not necessarily the definitions which we might assume.

3.1.1. Entity

An object of interest in the system which will undergo or cause change during the system’s activity. — Sutcliffe

An object which is involved in a time-ordered set of actions. the time ordering of entities is modelled with entity structure diagrams which express the time-ordering constraints. Entities become model processes within the system. — Sutcliffe

An object in the real world which participates in a time-ordered set of actions. — Jackson

A person, organisation, or object that performs or suffers an interesting time sequence of actions. — Cameron

3.1.2. Action

An event which happens to an entity. — Sutcliffe

Something that happens which the system has to respond to. Actions / events are atomic and cause a response in the system and an update to the information stored within the system. — Sutcliffe

An event in which one or more entities participates by performing or suffering the actions. — Jackson

An event in the world that forms the subject matter of the system and about which the system must produce or use information. — Cameron

3.1.3. Examples

Given the above definitions, examine the following list of examples and decide whether you agree or not.

The following are candidate entities:

• A customer in a banking system;
• A supplier in a stock control system;
• A sales clerk in a sales system;
• An application in a student admission system;
• A purchase order in a purchasing system;
• An examinations board in an examination system;

1 Note the object oriented flavour of these definitions

2 That is, they occur in an instant, or short period of time.
• An invoice in a sales order processing system;
• A switchboard in a telephone system.

The following are candidate actions or events:
• Open an account, make a payment, make a withdrawal;
• Receive an order request, satisfy an order request;
• Vet a sales order, establish credit limit;
• Be received by admissions tutor, be rejected;
• Be matched with sales, be priced, be despatched;
• Be convened, take place, report findings;
• Receive incoming call, transfer to extension.

3.2. Finding Entities From a Statement of Need

The entity analysis phase of the modelling state is all about obtaining a clear understanding of the system requirements and hence has as a starting point a very rough specification which may be no more than a simple statement of need. During this phase the systems developer will elicit information from the user community in order to achieve this objective.

3.2.1. Guideline For Discovering Entities

As discovering entities is considered to be difficult, Sutcliffe recommends a number of guidelines:
• Look for real world objects with which the system will have to interact;
• Ask What is the main thing for which the system exists?
• make a list of phrases which happen in the system, e.g. in a bank: customer borrows money. From these phrases identify common nouns and list these as potential (or candidate) entities;
• Ask What (or who) are the principle things (or people) that are necessary to solve the problem? Take care to omit superfluous detail;
• Look for all the jobs or tasks that people perform and the documentary evidence of their work.
• Look for the mechanisms inherent in the system and the operations which must be performed to operate the system.

3.2.2. Refining the Candidate List of Entities

Using the above guidelines will allow the provision of a candidate list of entities which will be worthy of further refinement. This refinement takes place throughout this stage and will involve:
• Identifying duplicate entities (different names but the same object).
• Removing entities which are outside the system boundary.
• Eliminating phantom entities i.e. those without any actions to perform or suffer.
• Matching actions to the entities and describing their time-ordering.
• Identifying marsupial entities these are hidden entities that can be identified by examining situations where there are interleaving common events.
3.3. Describing Actions

Actions must occur in a short period in time (since they will be modelled as atomic events), so, when identifying and describing actions eject those which are vague or imply continuous action. Also reject at this stage any actions which imply output or responses in the real world—actions in the modelling stage are those which respond in some way to input messages.

3.3.1. Guidelines for Determining Actions

We draw up lists of actions with a sentence or two of concise English text which lists the data which are necessary for the action. The following guidance is given by Sutcliffe:

- Decide on the external events which happen in the real world. e.g. a customer orders some goods.
- Define how these events are communicated to the system as input. e.g. a telephone call is made.
- Identify the system inputs (these are action attributes). e.g. a (verbal) order.
- Define actions which are necessary to respond to the messages contained within action attributes. e.g. the order is confirmed, the goods are despatched etc.

3.3.2. Action Attributes

Action attributes are the answer to the sensible question we might reasonably ask about action (or event) occurrences. For example, if we are told that a purchase event has occurred, we might ask What was purchased?; When was it purchased?; How much did it cost? Giving us the attributes: item description; purchase details; date of purchase; price. We would need to decide exactly what we mean by price, date and so on. There is a potential for overlooking attributes by not asking the right questions here—take care!

3.3.3. Allocating Actions to Entities

After refining the list of actions and the list of entities as best we can in isolation, we allocate the actions to the entities. In so doing we might discover that some of our candidate entities are not to be included because they have no actions and hence no time-ordering. At the same time we are concerned with the order of events (actions) and make a preliminary action sequence to depict each entity's life history. Later we will show this by means of the process structure diagram. If we get our time-ordering wrong here, then future stages will also be incorrect. It is vital to spend a large amount of time and effort in the initial modelling state.
3.4. XYZ Warehouse Co.—A Case Study

Customers order products from the company by telephone, mail or personal visit to the warehouse. Each product required must have a separate order, which is acceptable because of the nature of the product. Orders may be cancelled or the quantity and/or the delivery date may be changed. A clerk deals with the customers and is responsible for allocating stock to outstanding orders. He checks the availability of stock by telephoning the warehouse. It is not required to develop the Company purchasing side of the business, only the sales system handling customer orders.

3.4.1. Finding Entities

From the above description, and using the given guidelines, produce a list of candidate entities. State your assumptions and/or ask for further clarification of the mechanics or personnel involved.
Possible entities might be:

- Customer
- Clerk
- Order
- Product
- Warehouse

Compare this list with your own (the above is not necessarily the ultimate list)

3.4.2. Finding Actions

From our case study description and any other information you have elicited, use the guidelines to produce an action list with a sentence or two to describe the action in terms of its attributes.

Here is one to start:

**PLACE** A customer places an order with the company for allocation and delivery.
Possible Actions might be

**PLACE**  A customer places an order with the company for allocation and delivery.

**AMEND**  A customer changes the quantity of the order.

**CANCEL**  A customer cancels an order.

**DELAY**  The clerk has to delay an order because of unavailability of the required stock.

**ALLOCATE**  Stock is allocated to an order.

**DELIVER**  The requested stock is delivered to the customer.

### 3.4.3. Deriving Action Attributes

Using the *Who?*, *What?*, *How Much?* type questions provide an attribute list for each of the above identified actions. Do not go into detail of size and data type at this stage.

**PLACE**

---

---

---

---

**AMEND**

---

---

---

---

**CANCEL**

---

---

---

---

**DELAY**

---

---

---

---

**ALLOCATE**

---

---

---

---

**DELIVER**

---

---

---

---
Chapter 3 - The JSD Modelling Stage

The resultant list of actions and their attributes might be:

**Action PLACE**
- **Summary**: The customer places an order
- **Attribute List**:
  - PRODUCT IDENTIFICATION
  - QUANTITY
  - REQUIRED DELIVERY DATE
  - CUSTOMER IDENTIFICATION
  - DATE ORDER PLACED
  - ORDER NUMBER

**Action AMEND**
- **Summary**: The customer amends the quantity and/or required delivery date.
- **Attribute List**
  - ORDER NUMBER
  - NEW QUANTITY
  - NEW REQUIRED DELIVERY DATE

**Action CANCEL**
- **Summary**: A customer cancels an order.
- **Attribute List**
  - ORDER NUMBER

**Action DELAY**
- **Summary**: An order is delayed because of insufficient stock.
- **Attribute List**
  - ORDER NUMBER
  - NEW REQUIRED DELIVERY DATE

**Action ALLOCATE**
- **Summary**: An order has product stock allocated to it.
- **Attribute List**
  - ORDER NUMBER
  - QUANTITY
  - PRODUCT IDENTIFICATION

**Action DELIVER**
- **Summary**: An order is delivered to the customer.
- **Attribute List**
  - DATE DELIVERED
  - DATE OF LEAVING
  - PRODUCT IDENTIFICATION

3.4.4. Allocating Actions to Entities

Using the candidate list of entities and the above action list attempt to allocate actions to entities in the order you would expect them to occur.

**CUSTOMER**

**CLERK**

**ORDER**

**PRODUCT**
What do you conclude in respect to the candidate list of entities?

3.4.5. Final Entity List

The list of entities has been reduced to four, because they are the ones within the system with time-ordered actions. The list with actions in approximate time-order is:

Entity **ORDER**
Summary This entity models the life of an order, from when it is first placed to the delivery of the stock or the cancellation of the order.
Action list
- Action **PLACE**
- Action **AMEND**
- Action **DELAY**
- Action **ALLOCATE**
- Action **ALLOCATE**
- Action **DELIVER**
- Action **CANCEL**

Entity **CUSTOMER**
Summary This entity models the activities of a customer in respect of all his activities in relation to dealing with XYZ warehouse.
Action list
- Action **PLACE**
- Action **AMEND**
- Action **CANCEL**
- Action **DELIVER**

Entity **CLERK**
Summary This entity models the activities of the clerk in dealing with orders.
Action list
- Action **DELAY**
- Action **ALLOCATE**

Entity **PRODUCT**
Summary This entity models the product in respect of its availability for delivery.
Action list
- Action **ALLOCATE**
- Action **DELIVER**

Note the common actions for the above entities; e.g. the clerk allocates some stock, while the product suffers the allocation. We may, if we choose, distinguish between performing and suffer-
ing actions; e.g. the actions deliver might be redefined as receipt for the cus-
tomer and removal (of stock) for the product.

We might also note at this time that we have taken a simple view of customer
and order and that it is almost certainly going to be the case that a customer
could have more than one order outstanding, and hence the actions must be
very clearly defined.

We shall return to this thought later.

3.5. Producing Entity Structures

A reminder of structure diagram conventions.

![Diagram of a sequence](image)

**Figure 3.1: A Sequence**

The component A is a sequence of B followed by C followed by D. Or, in entity
structures, the entity A is a sequence of action B followed by action C followed
by action D.

![Diagram of a selection](image)

**Figure 3.2: A Selection**

The component E is a selection of F or G or H. Or, in entity structures, the entity
E is a selection of action F, or action G, or action H.

![Diagram of an iteration](image)

**Figure 3.3: An Iteration**

The component J is an iteration of K. Or, in entity structures, the entity J is an
iteration of the action K.

3.5.1. What Are We Showing With Entity Structures
Essentially, for each entity (represented by the root box) we depict the actions suffered or performed by that entity in time-order. Hence the notion of a sequence is very important. When actions are repeated, then the iteration construct is useful; and when there are alternative actions in the life of an entity, the selection is used. As with JSP diagrams, we can construct entity structure diagrams with a number of components nested within other components. Hence, in the following diagram, the entity \( A \) is a selection of the action \( B \) followed by the action \( C \) followed by the action \( D \); the action \( C \) is an iteration of the action \( E \); and the action \( E \) is a selection of either the action \( F \) to the action \( G \).

![Figure 3.4: Sample entity structure](image)

Note that, in order to obey the rules for construction of entity structure diagrams or to clarify/simplify the structure, it may be necessary to add high level boxes. For example, in the diagram above, the box \( E \) may not be a direct action in our list, but it might be necessary in order to show an iteration of either \( F \) or \( G \). Again, this is just as in JSP with the introduction of body boxes.

### 3.5.2. Backtracking in Entity Structures

An entity structure should model the whole life history including all possible events and sequences of events. In some circumstances, entities indicate a number of selection constructs, often deeply nested, in an attempt to deal with uncertainty; e.g. a succession of actions only occurs under a certain combination of circumstances and this is only determined when the system input data is applied. This may lead to untidy and/or complex structures which can be avoided by using the technique of backtracking. Often, the technique is used to indicate a premature end to the actions of an entity. The JSP notions of *posit, admit* and *quit* are used. For example:
3.5.3. Marsupial Entities

There is no way of showing concurrency in a single structure diagram. If we have a customer who has a bank account it is easy to model the opening, paying in, withdrawing and closing events with some certainty of their time-ordering. But when he is allowed to have a number of accounts, then it can get confusing in the modelling process because of the potential for concurrent actions of the customer in respect of different accounts. The solution is to identify the account as a sub-entity which can exist on its own (an emerging marsupial) and model accordingly.

Marsupials will share common actions with their parent entities\(^3\) and often can be recognised as repeating groups within entities (Note that removing such repeating groups is in effect performing a type of analysis to first normal form).

3.6. How to Produce Entity Structure Diagrams

Sutcliffe suggests the following list of heuristics as a guide to producing entity structure diagrams.

- Start by ordering actions in a sequence from the first to occur in the entity’s life on the left-hand side of the diagram to the last action placed on the right hand side.
- Add any choices and repetitions to the diagram, and re-order to obey diagramming conventions. This will add selections and iteration to the diagram and will naturally add more layers or depth to it as the linear sequence is changed.
- Add any higher-level components which may be needed to make the structure more comprehensible. This may not be necessary; in an entity structure diagram which consists of a flat sequence of actions, however, grouping will

---

\(^3\) c.f. Inheritance in the object oriented paradigm.
help to make the diagram easier to read by adding hierarchical structure, e.g. opening actions - middle actions - closing actions.

- Criticise each action definition to ensure that it is neither too trivial nor too large. If actions are unsatisfactory either subdivide them or eliminate them from the diagram.

We might add:

- Examine the resultant structure to look for uncertainty in the actions of iterations or selections. Particularly look for nested selections which have been produced in order to cater for premature end situations. In these cases attempt to re-draw the diagram using appropriate posit, admit and quit boxes.
- Look for repeating groups (i.e. high level iterations) which could give rise to marsupial entities, and separate them off into sub-entities.

3.6.1. An Example

Given the following entity description for a book in a library system, let us construct the process structure diagram according to the above guidelines.

**Entity** BOOK

**Summary** This entity models the life of a copy of a book, from acquisition by the library through to disposal by either sale or inter-library swap.

<table>
<thead>
<tr>
<th>Action list:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action <strong>ACQUIRE</strong></td>
</tr>
<tr>
<td>Action <strong>CLASSIFY</strong></td>
</tr>
<tr>
<td>Action <strong>LEND</strong></td>
</tr>
<tr>
<td>Action <strong>RENEW</strong></td>
</tr>
<tr>
<td>Action <strong>RETURN</strong></td>
</tr>
<tr>
<td>Action <strong>SELL</strong></td>
</tr>
<tr>
<td>Action <strong>WITHDRAW</strong></td>
</tr>
</tbody>
</table>

3.6.2. Order the actions

First we order the actions in a sequence from the first to occur in the entity’s life on the left-hand side of the diagram to the last action placed on the right-hand side. This gives us:

![Diagram of time-ordered actions](image)

Figure 3.6: Time-ordered actions

3.6.3. Add selections and iterations

Next, we note that a book may be lent many times during its useful life and when it is on loan it is not necessarily renewed. Also, when it is disposed of at the end of its life it is by either selling or withdrawing. Hence, we now add selections and iterations to the diagram and re-order to obey diagramming conventions.
Chapter 3 – The JSD Modelling Stage

3.6.4. Search for Uncertainty

Finally, we examine the resultant structure to look for uncertainty in the actions or iterations or selections and conclude that we do not need to use back-tracking.

3.7. Case Study—XYZ Warehouse

Let us now produce an entity structure diagram for our warehouse case study. Recall, the actions for a customer are: PLACE, AMEND, CANCEL and DELIVER. But we know that a customer may cancel an order or have it delivered; and that he may or may not make amendments. This could give rise to the process structure diagram:

We have added some higher-level components in order to obey the construction rules, but we do not need any more to make the structure more comprehensible.

However, we remember that a customer could have a number of orders outstanding at any time; hence, the above model is inadequate to distinguish between one order and another. We could impose an arbitrary limit on the number of orders allowed per customer and produce a structure based on that limit with selection constructs to identify which order is being referred to. But this would...
be hopelessly complicated. So we produce the realistic structure for customer and also its marsupial entity which is for a customer order.

![Figure 3.9: Re-drawn Customer](image)

The entity **PRODUCT** has just two actions: **ALLOCATE** and **DELIVER**. The structure of this entity is simply an iteration of actions which may be **ALLOCATE** or **DELIVER**. Note that no time-ordering may be imposed on these actions because for any given produce the allocation and delivery (or issue) of stock does not necessarily alternate because a product may be the subject of a number of orders with interleaving allocation and delivery dates.

![Figure 3.11: Product PSD](image)

Close inspection of the probable role of this entity in our model casts doubt as to its inclusion. Because we are only concerned with the sales system our contact with product is confined to an inspection of a product's state (clerk phones up to check availability). We may conclude that we can omit this entity from our model. We will return to this when we discuss the automation of the clerk’s activities.
Chapter 3 – The JSD Modelling Stage

For the **CLERK** entity, construct the appropriate entity structures, noting that a clerk may deal with many orders.

![Figure 3.12: Clerk PSD](image)

And its marsupial:

![Figure 3.13: Clerk's marsupial entity](image)

We now have marsupials both relating to order (this has arisen because of common actions). The entity **ORDER** can now be described using a structure diagram. It will in effect be a composite of the two marsupials. Recall that the actions of **ORDER** are **PLACE, AMEND, CANCEL, DELIVER, DELAY, ALLOCATE**.

Using all the above information produce a process structure diagram for the entity **ORDER** based on **CUSTOMER ORDER** and **CLERK ORDER**.
A possible solution with the imposed rule that a customer cannot amend or cancel an order after it has been allocated is given. (Further investigation would reveal whether this imposition is realistic).

![Customer-Order entity](image)

**Figure 3.14: Customer-Order entity**

### 3.8. Producing Structure Text

Structure text (similar to Schematic Logic in JSP) is an alternative specification method to process structure diagrams. It may be used in parallel with structure diagrams or instead of them. The syntax for the three constructs: sequence, selection and iteration are represented in structure text by:

#### 3.8.1. Sequence

```
Entity name Seq
   Action_name 1;
   Action_name 2;
   ...
   Action_name n;
Entity name End.
```

The entity is a sequence of \( n \) actions.

#### 3.8.2. Selection

```
Entity name Sel (first condition)
   Action_name 1;
Entity name Alt (second condition)
   Action_name 2;
   ...
Entity name Alt (nth condition)
   Action_name n;
Entity name End.
```

The entity is a selection of \( n \) actions.
3.8.3. Iteration

<table>
<thead>
<tr>
<th>Entity name Iter (while condition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action name 1;</td>
</tr>
<tr>
<td>Entity name End.</td>
</tr>
</tbody>
</table>

The entity is an iteration of action 1 while the condition remains True.

3.8.4. Putting it Together

Refer back to the process structure diagram on page 26; the equivalent structure text is:

```
A Seq
 B;
 C Iter (while condition 1) 
    E Sel (condition 2) 
    F;
    E Alt (condition 3) 
    G;
    E End
 C End;
 D;
 A End.
```

Notice how the indentation reflects the hierarchy of the diagram. Structured indentation makes a program easier to read and helps the programmer who follows during future maintenance.

Using the process (entity) structure diagrams already produced, let us produce equivalent structured text for the four entities CUSTOMER, ORDER, PRODUCT and CLERK.

Here is the first one:

```
Customer Iter (while still a customer)
 Action Sel (if a new order is placed)
    Place;
 Action Alt (if an order is amended)
    Amend;
 Action Alt (if an order is delivered)
    Deliver;
 Action Alt (if an order is cancelled)
    Cancel;
 Action End
Customer End.
```

Now produce structure text from the entity structures given on pages 30 to 32 for CLERK, PRODUCT and ORDER.
3.8.5. Solutions

This gives us:

<table>
<thead>
<tr>
<th>Clerk Iter (while employed in this capacity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Sel (if an order is to be delayed)</td>
</tr>
<tr>
<td>Delay;</td>
</tr>
<tr>
<td>Action Alt (if stock is allocated)</td>
</tr>
<tr>
<td>Allocate;</td>
</tr>
<tr>
<td>Action End</td>
</tr>
<tr>
<td>Clerk End</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product Iter (while used by warehouse)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Sel (if stock is allocated)</td>
</tr>
<tr>
<td>Allocate;</td>
</tr>
<tr>
<td>Action Alt (if stock is delivered)</td>
</tr>
<tr>
<td>Deliver;</td>
</tr>
<tr>
<td>Action End</td>
</tr>
<tr>
<td>Product End</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order Seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place;</td>
</tr>
<tr>
<td>Delays &amp; Amends Iter (while order not dealt with)</td>
</tr>
<tr>
<td>Action Sel (if an amendment)</td>
</tr>
<tr>
<td>Amend;</td>
</tr>
<tr>
<td>Action Alt (if a delay)</td>
</tr>
<tr>
<td>Delay;</td>
</tr>
<tr>
<td>Action End</td>
</tr>
<tr>
<td>Delays &amp; Amends End</td>
</tr>
<tr>
<td>Deliver or cancel Sel (if completed)</td>
</tr>
<tr>
<td>Complete Seq</td>
</tr>
<tr>
<td>Allocate;</td>
</tr>
<tr>
<td>Deliver;</td>
</tr>
<tr>
<td>Complete End</td>
</tr>
<tr>
<td>Deliver or cancel Alt (if cancelled)</td>
</tr>
<tr>
<td>Cancel;</td>
</tr>
<tr>
<td>Deliver or cancel End</td>
</tr>
<tr>
<td>Order End</td>
</tr>
</tbody>
</table>

3.9. Exercises

1. From the following brief description, separate out phrases which indicate the major nouns (to identify entities) and the verbs (to identify actions). From these:

   i. Produce an action list to the entities you have discovered;
   ii. Allocate the actions to the entities you have discovered;
   iii. Produce entity (process) structure diagrams for the entities;
   iv. Produce the equivalent structure text.

A certain machine is used to either make or finish components. The machine is set up at the start of each day by an operator. When the machine is used for making new components it is loaded with raw materials; when used for finished components that have already been made it must obviously be loaded with the unfinished components. A product must be ordered before it is made and it is put into a warehouse after it has been finished and inspected before being delivered. If it is found faulty at inspection, it may be re-finished or rejected. If re-finished, it is assumed to be fit for delivery.
1. The life history of the entity student contains the following possible events:
   ii. applies for a course;
   iii. enrols for the course;
   iv. progresses to the 2nd year;
   v. turns down an offer of a place;
   vi. withdraws during the 1st year;
   vii. withdraws during the 2nd year;
   viii. passes exams into 2nd year;
   ix. fails exams in 2nd year;
   x. accepted for course;
   xi. rejected for course.

   - Draw a process structure diagram for the student entity without using backtracking.
   - Produce the equivalent structure text for the student entity.
   - Redraw the diagram using backtracking.
   - Produce the equivalent structure text for the backtracking version.

3. Competitors for a motoring speed trial start from a single grid position at different times and complete four laps which vary in length as different parts of the circuit are brought into use. As each car passes the check-point/finish a radio signal relays car identification and time of day. Marshals signal the starting time as each competitor sets off. Points are scored as a function of time taken. On the assumption that each record is sent to a single queue and that all competitors who start also finish, produce an initial model for a system which will calculate each competitors final score for public display.
4. The Network Stage—Initial Model

4.1. The System Specification Diagram (SSD)

In the previous chapter we showed you how to identify actions within the problem domain, associate those actions together within entity groupings and then place actions within control structures and in a time ordered sequence to produce entity life histories. These are also known as entity models, process structure diagrams, model processes or program structures.

These model processes may be likened to individual programs, and as such, have input and output. A normal computer system is made up of many such programs. Their input and output data going to and from many sources and sinks, often other programs. The development process therefore needs to be extended to define how these processes communicate with each other.

A logical model of the whole system needs to be built up: a network of concurrent communicating sequential processes. A new form of notation is required by which concurrency may be expressed. This notation allows us to construct a network model, known in JSD as a system specification diagram (SSD). The SSD is significantly different from the previous hierarchical tree diagrams; the SSD is a variation of the node-arc type of diagram.

The SSD allows us to create a logical map of the system, showing how the individual processes communicate with each other and the real world. To start with the SSD assumes that all processes are concurrent, that is, they are capable of running independently of each other, halting only when they require input that is unavailable.

Clearly, our finished system is unlikely to be able to run in such an environment, one which allows for individual processors for individual processes. Also even if we did have such an environment, it would be undesirable for some processes to run unchecked. For example, what is the point of a report writing process that writes reports, one after the other, as fast as possible, if the report is only required once a week?

As we progress through the next few chapters the network diagram will first be created as a purely uncontrolled concurrent model. It will then be progressively refined/modified until we have a network model which may be implemented on the desired environment, which includes consideration of the target hardware.

The SSD initially comprises the following components:

- **Model processes** which are made up of entities and their time ordered actions;
- **Functional processes** which are added at the elaboration phase (see later);
- The means by which the model processes and function processes communicate or are interconnected:
  - **Data stream** connections which are FIFO message passing;
  - **State vector** connections which are read-only inspections of attribute states.
4.2. Basic Notation

4.2.1. Process

The notation for a process is a rectangular box containing the process name. When we start constructing the SSD we are only interested in the individual model processes as a single item. We do not need to know about their internal structure. Therefore, on the SSD we can represent the entire model process by the use of a single symbol—the rectangular process box. This is the root component of the model process hierarchical tree structure.

![Customer process](image)

N.B. A more formal JSD notation would be:

![Real World Model Process](image)

4.2.2. Data Stream

The notation for a data stream connection is a circle containing the data stream identification:

![Data Stream](image)

4.2.3. State-Vector

The notation for a state-vector connection is a diamond containing the state-vector identification:

![State Vector](image)

4.3. Some Examples of SSD Components and Connections

![Customer process](image)

Here, one CUSTOMER process is connected to one BANK process by a data stream c written by CUSTOMER and read by BANK.
Here, one ENQUIRY-B process is connected to one BALANCE process by ENQUIRY-B directly inspecting the state-vector of BALANCE. Consider which process is inspecting the state of which process, which way the data is flowing and the direction the arrows on the connections are pointing.

Here, many CUSTOMER processes are connected to many BALANCE processes by data streams c written by the CUSTOMER processes and read by the BALANCE processes.

Here, one ENQUIRY-B process is connected to many BALANCE processes by ENQUIRY-B directly inspecting the state-vectors of the BALANCE processes.

4.4. Data stream Connection

Jackson’s definition of a data stream is an ordered set of records or messages by which two processes communicate, the records being read by one process in the order in which they are written by the other.

A data stream is then, a message channel which acts like a first-in/first-out buffer. Data streams make up the majority of connections in the SSD and are the only way of receiving input from the real world. Two processes are involved, the reader process and the writer process which takes an active role in the connection; the two processes are said to be tightly coupled.

The writer process actively writes a stream of data (either individual data items or records of data) to the reader process. Because both processes are concurrent, the writer process may reach a point where it is required to write a piece of data before the reader process is ready to receive it. In the data stream connection every piece of data must be processed, so if the reader process is not ready to receive the piece of data then it must be placed in a buffer (infinite buffering is assumed at this stage). Not only must every piece of data be processed, they must also be processed in the order in which they are written, i.e. in sequence. The buffer must therefore reflect this and store the unread data in sequence.

There is the problem of what happens if the reader process is ready for the next piece of data and the writer process is not in a position to transmit it. In this case, with the buffer empty, the reader process can do nothing. It cannot proceed until it has some data to process therefore if must halt. This is the fundamental difference between state vectors and data streams. The data stream re-
relationship means that the writer process can exert control over the reader process. By deliberately not making data available, it can prevent the reader from proceeding.

The concept is well illustrated by two people washing and drying dishes with the constraint being that the dishes must be dried in the order that they are washed. The one who washes the dishes controls the whole operation because if he works fast, the other person needs to work fast to keep up (or else let a large backlog of dishes pile up in the dish rack, that is, the buffer); and if he stops then the drier stops when there are no more dishes to dry.

Data streams are used to connect processes as a means of communication when all changes of state of an attribute are required (for example, a bank statement is a report of all changes of state). In contrast, a state-vector is used when a snap-shot of the current state of an attribute is required (for example, a balance enquiry).

The timing of the read-write relationship isn’t closely defined at this stage because the processes are asynchronous; hence the possibility of blocking the reading process.

The type of message in a data stream is not closely defined; hence variable or fixed length records of varying types may be defined. The contents of the messages are determined by the actions described for the process (entity).

Read and write instructions may be added to the process structure diagrams as elementary operations and may be placed as Read(data stream) and Write(data stream) statements in the structure text.

4.4.1. Rules for the Placement of Reads and Writes

The basic rule for correct placement of reads and writes is:

- Reads must be achieved before the actions which require data; they are placed immediately after an action to replenish the input with the next message (the read-ahead rule).
- Writes are placed after actions which produce data.

4.4.2. Example

Given the following fragment of an SSD:

![Diagram](image-url)
Consider the structure text for the two processes:

**Proc-a**

```
SEQ
Action a-1;
Write A1 to ds-T;
Proc-a-body ITER (forever)
  Choose-a SEL
  Action a-2;
  Write A2 to ds-T;
  Choose-a ALT
  Action a-3;
  Write A3 to ds-T;
  Choose-a END
Proc-a-body END
Proc-a END
```

**Proc-b**

```
SEQ
Read ds-T;
Action b-1;
Read ds-T;
Proc-b-body ITER (while A2 or A3)
  Choose-b SEL (A2)
  Action b-2;
  Read ds-T;
  Choose-b ALT (A3
  Action b-3;
  Read ds-T;
  Choose-b END
Proc-b-body END
Proc-b END
```

Imagine an infinite buffer into which the messages A1, A2, A2, A3, A2, A3, A3, A3, … are written by **Proc-a**. Use this information to demonstrate to yourself how **Proc-a** controls **Proc-b** given that there is a data stream connection between the two concurrently executed processes.

### 4.4.3. Data stream Connection Characteristics Summarised

Consider the following SSD:

![Data Stream Diagram](image)

- The initiative lies with the process (W) that writes the data stream to determine the content and order of the records of the data stream (ds). This can be deduced by considering the behaviour of W alone without considering the reading process (R) at all.

- R is forced to read every record written by W. There is no mechanism by which R can choose to skip records of ds, or to read them in a different order from their writing by W.

- If, for any reason, execution of R is slowed down or stops temporarily, records of ds wait in a buffer until R continues execution and reads them. There is no mechanism by which they wither away, or are read by some other process, or are overwritten by later records.

- If execution of R reaches a Read ds operation, and there is no record available in the buffer (because R has already read all the records that W has written so far), R is said to be halted or blocked. Execution of R cannot continue until a record becomes available and the Read ds operation can be completed.

- Because the data stream is buffered, W is not blocked when it reaches a Write operation, irrespective of the number of records previously Read by R. The buffer is considered to be of unlimited capacity; the Write operation in W is completed by the addition of the record to the buffer, and W continues execution.

### 4.5. Merging Data Streams

When two or more data streams are read by the same process they are said to
be merged. This means that the mechanism the reader employs to read data from one process can be used to read data from another process.

Two or more processes write data to the reader process. Normally each data stream has its own buffer and the reader process must read and process the next item of data in that buffer. After reading all the data from one buffer the reader process then moves to the next buffer and reads all the data from that buffer and so on.

If the data stream is being used to exert control then clearly it must utilise its own buffer, so that by not writing to the buffer, the reader process may be halted. However, often this is not the case. If there is no requirement for incoming data to be processed in strict order, but merely in the order in which it is written, then rough merge data streams may be used. The other form of merge is the fixed merge; we will consider these below.

4.5.1. Rough Merge

In rough-merged data streams, two or more data streams share the same buffer. The writer processes run independently, writing data to the buffer whenever they wish. The data still arrives at the buffer in the order that it is written but it may be interleaved with the sequential data from other data streams. However, the reader process however only receives data from a single buffer. It must still read data FIFO but it does not matter which particular writer’s data is read next.

The data stream rules still apply: when the buffer is empty the reader process must halt. The difference from a fixed merge case being that an individual writer process data stream can dry up without halting the reader process. Halting can only happen when all writer data streams dry up.

The SSD below shows rough merged data streams in a way which reflects the view of the reader i.e. the reader only sees the one incoming data stream. It is unaware that the data it is reading originates from more than one writer process.

Here, the buffer collects data form the input data streams from DEBIT and CREDIT, merges that data in an interleaving FIFO order and outputs a single data stream containing data from both DEBIT and CREDIT to the READER PROCESS.

If the sequence of message arrival is unpredictable, for example when payments and withdrawals can come in any order, then we would use a rough merge which could be represented by the following structured text:
The disadvantage of the above is that withdrawals are favoured before payments. For instance, if withdrawal and payment records arrive simultaneously, then withdrawals will be processed first. (If we were a bank, we might actually wish this to be so.) If we wish to define priorities explicitly then we could introduce a specific process to prioritise the message handling, represented on an SSD as follows:

```
Process Get-payments-&-w/d Seq
  Possible record Sel (withdrawal data stream not empty)
    Read (withdrawal);
    Action for Withdrawals;
  Possible record Alt (payments data stream not empty)
    Read (payments);
    Action for Payments;
  Possible record End
Process Get-payments-&-w/d End
```

In effect, this process **MESSAGE HANDLER** gives us a form of guaranteed fixed merge from indeterminate input order.

### 4.5.2. Fixed Merge

The name is slightly misleading since data isn’t actually merged but remains formally separate. It is the more usual method of handling incoming data flows into a process.

The fixed merge is the **only** possible mechanism for handling the arrival of state vector data at the reader, because no buffers are involved. In the fixed merging of data flows the reader process processes each input in turn. The order of processing is determined by the process structure and is not implied on the SSD. Below is an example:

Where data streams are involved in a fixed merge the reader process must alternate between writer processes and halt if no data is available, regardless of whether the other processes have data that could be read. Unless the processing of data streams is carefully constructed there is a danger of deadlock. consider the following example:
The reader process requires data from both DS1 and DS2 before it can produce data for DSR. Write Process 2, which is responsible for producing data for DS2, can only function if it receives data from DSR. It is not difficult to comprehend how deadlock can occur in situations like this.

Therefore, if the sequence of message arrival is known, (e.g., if debit messages always followed credit messages in their own data streams) we can use a fixed merge which can be represented in structure text as follows:

```plaintext
Process Get-credits-&-debits Seq
   Read (credit stream);
   Action for credits;
   Read (debit stream);
   Action for debits;
Process Get-credits-&-debits End
```

Finally, it should be pointed out that rough and fixed merges may be mixed so that they flow into the same reader process.

### 4.6. State Vector Connections

Jackson defines a state vector as **the local variables of a process, including the text pointer**, i.e. life history has a pointer.

An inspector process may inspect the state vector of another process at any time because it is always available to the inspector process. This is important because it means that the inspector will never have to wait for data, because the inspector neither has to wait for data, nor does it need all the changes of state to be provided, the state vector connection forms a loosely-coupled process connector. There is no concept of the writer actually writing any data. The state vector itself consists logically only of the internal variables of the process whose state will vary with the normal activities of the program.

A state vector connection then, is like an inspection of the local variables (data) of one process by another process.

State vectors are read by **Get(sv)** instructions which are entirely the responsibility of the inspecting process.

Timing of **Get(sv)** instructions can be very important because it may be necessary to ensure that the snapshot of a process's state is taken at a particular point in time.

The arrow heads denote the direction of the data and must therefore, both point in the same direction. There can only ever be one process as a writer and one process as an inspector in a state vector connection. If more than one process is inspecting the state vector of a writer process then there must be a separate flow for each inspector process as shown below.
This raises another issue regarding naming the state vectors and more importantly what the state vector consists of. In the example above, each diamond represents the state vector of the same process. It would therefore seem logical to give them all the same name. There is a problem with this, namely one of different views of data. Each inspector process is only interested in specific segments of the writer’s state vector i.e. only the values of certain variables are required. Therefore each diamond should not only reflect the identity of the writer process but should also exhibit uniqueness.

State vector connections are common when functional processes are added to the SSD, for example when reports and displays are required.

### 4.6.1. State Vector connection characteristics summarised

Consider the following fragment of an SSD:

- The initiative in the communication lies entirely with the reading process \( (R) \). There is no operation in the inspected process \( (I) \) to cause its state vector to be inspected by \( R \).
- The \( \text{Get(sv)} \) operations executed by \( R \) are determined entirely by \( R \) itself. \( I \)’s state vector will take a succession of values determined by the structure of \( I \), but these values will not affect \( R \) except to the extent that \( R \) executes \( \text{Get(sv)} \) operations.
- If execution of \( R \) is slowed or stopped temporarily, the values of \( I \)’s state vector may change many times before \( R \) gets another value of \( I \)’s attributes state; in other words values may be missed between \( \text{Get(sv)} \) operations.
- The \( \text{Get(sv)} \) operation does not cause blocking of \( R \) or \( I \).

### 4.6.2. Difference between state vectors and data streams

The differences between state vectors and data streams is much like the difference between a simple clock and one with an alarm. Suppose a man needs to be up at seven o’clock to go to a job interview. He has an alarm clock which he sets before he goes to bed. He then sleeps through the night until the clock’s alarm wakes him up. This is analogous to the data stream connection. The man is the reading process and the clock the writing process. The man is halted (asleep) waiting to read data from the writing process. At such time as the writing process is ready to write data to the data stream (i.e., seven o’clock in this case) it
does so thus causing the reading process to reactivate (or the man to wake up).

The state vector may be likened to the clock without the alarm. As the man now has no way of knowing when it is seven o’clock, he must continually inspect the state of the clock process (look at it) to see what the time is. If it is seven o’clock then he can get up, otherwise he should stay in bed. Notice this time that the onus is now on the inspecting process (the man) to obtain the data he wants (the time) at the correct moment (seven o’clock). If he looks at the clock too early then he will have to check it again later. If he looks at it too late (perhaps he fell asleep in the middle of the night) then he will have missed the important item of data (and be late for the interview).

4.7. The Initial Model Phase and the Case Study

In a realistic system we would naturally wish to automate the allocation and delaying of orders. This would have the effect of making these events system’s functions which in turn would mean that we do not need to model the clerk as an entity. This in turn means that we must reconsider the remainder of our model.

First the entity CUSTOMER; there is no change here unless we envisage our automation having an impact upon the behaviour of the customer.

Next, the entity PRODUCT; we were a little unhappy about the inclusion of this entity in our model, but if we are to automate the clerk’s activities then the availability of a product is of concern to us. We must, therefore, model the behaviour of PRODUCT as abstracted from the point of view of its use in the sales system., as having the single action AVAILABLE: (the state would either be: TRUE or FALSE).

**Action AVAILABLE**

- **Summary**: A quantity of a product is available for allocation to orders.
- **Attribute List**:
  - PRODUCT IDENTIFICATION
  - QUANTITY

**Entity PRODUCT**

- **Summary**: This entity models the product in respect of its availability for allocation.
- **Action List**:
  - Action **AVAILABLE**

Next, the entity ORDER; because the events DELAY and ALLOCATE become part of the system function, they are not needed in the entity structure. Hence, the new structure for ORDER and its corresponding structure text are:

**Entity ORDER**

- **Summary**: This entity models the life of an order, from when it is first placed to the delivery of stock or the cancellation of the order.
- **Action List**:
  - Action **PLACE**
Chapter 4 - The Network Stage—Initial Model

4.8. The Initial Model

The real world of a customer is connected to the model process CUSTOMER by a data stream connection which we will call CUS-INP. The information from this data stream must also be used to connect the real world to the marsupial entity ORDER via the process CUSTOMER. The CUSTOMER process must write a data stream for this purpose, we will call it CUS-ORD. Hence we produce an SSD:

```
Order Seq
Place;
Possible Amends Iter (while order not dealt with)
  Amend;
Possible Amends End
  Deliver or cancel Sel (if delivered)
    Deliver;
  Deliver or cancel Alt (if cancelled)
    Cancel;
  Deliver or cancel End
Order End
```

The entity PRODUCT still causes us problems because really what we are trying to do is to periodically glimpse into another system (which is part of the sales system in the real world) to extract some information about the availability of the stock.

We have defined the action AVAILABLE as the only event of the entity PRODUCT. If the entity PRODUCT, in reality, is the stock record of the stock control system, then the action AVAILABLE can only occur when this record has been updated so that its quantity attribute is correct. The trick we need is to be able to synchronise our Get(sv) operations at the appropriate times. The simplest solution (but not necessarily the most convenient), is to model the PRODUCT such that each AVAILABLE action is preceded by an UNAVAILABLE action which represents the time when the stock control system is doing its work. The effect of this is to lock the stock control record from read access whilst it is being
written to. So we refine our model again:

**Entity** PRODUCT

Summary  This entity models the product in respect of its availability for allocation.
Action List:
Action AVAILABLE
Action UNAVAILABLE

```
Product
   └── State Pairs
      ├── Un-Available
      └── Available

Product Iter (while in catalogue)
State pairs Seq
Get (sv)
Unavailable;
Get (sv)
Available;
State pairs End
Product End
```
5. Network Stage Elaboration Phase

5.1. Adding Detail to the SSD

At the end of the initial model phase, the JSD specification consists of model processes (which are abstractions of real world entities) and their input and output data stream and state vectors. We have not been concerned with detail, but we have been very careful in trying to identify the major processes of interest within the system boundary that impact the real world. We have modelled, for example, a customer's activities as they represent the activities of the system that we are with; and we have modelled the communication between them. This is the core of the new system upon which we shall build and extend the specification.

We now add new functionality to cover the user requirements. This new functionality is of course merely a set of new processes or amendments to existing processes to incorporate the new functionality. However, whilst there is absolutely no structural difference between existing processes and the new processes, we still refer to them differently to show that these particular processes are not required for the system to fulfil its specification.

5.1.1. Input and Output Subsystems

Only after we are satisfied with the initial model do we concern ourselves with the other major system components, the input and output subsystems. In JSD the output subsystem provides information about the performance and status of the model processes. The input subsystem is responsible for the provision of timely and correct data from the real world to the computer system.

5.1.2. Stability of the Model Processes

Model processes are relatively stable parts of the system and should not be confused with functional processes which are derived from the activities (information processing and user-interface) of the system and can be subject to frequent change.

5.2. Adding Functional Processes

There are four types of functional process: embedded functions, imposed functions, interactive functions and filter functions. These will be reviewed in turn.

5.2.1. Embedded Functions

Embedded functions last a short length of time and may turn out not to be a function but a modelling oversight. This type of function alters an existing model process. They must not however alter the structure of the existing process.

Elementary operations may be added to existing model processes. For example, if it is required to output a new balance every time it has been updated, then for every occurrence of the event *update balance* we simply have an output operation which does not affect the time ordering of the model process at all. In such
cases we simply embed the new functional requirement if output balance directly within the model process by appending it as an elementary operation to the process structure diagram or the structure text.

5.2.2. Imposed Functions

An imposed function, in general, represents a larger processing task than an embedded function. They, like embedded functions, are a result of another function or a user requirement. Consider the embedded function which, by the addition of some actions, sends data for printing or screen output. Obviously there is a need for a report formatting process, this must be created—it must be imposed on the system. A new process must be added to the system.

Imposed functions are frequently used to inspect the state vector of their connected process(es) and are therefore usually of report writing or query screen nature. Since most state vectors are, in implementation terms, files, many database query programs are in effect imposed functions.

Imposed functions usually have to inspect the state vectors of many processes and usually in a strict order. Functions of a query nature tend towards the complex. Remember, query screens and so forth are rarely needed in order to fulfill the system specification. The requirements for them also tend to be rather fluid—a high maintenance area.

5.2.3. Some Examples of Impose Functions

Let us examine three different types of information request in respect of a system which models students’ attendance at lectures.

If we wished to examine periodically the pattern of attendance of a student (i.e. look at his attendance record over the period since we last looked), we might have a model process which outputs a data stream containing the attendance details which is rough merged with a user request data stream as input to a functional process which produces the required output. So we have:

![Diagram of Continuous Flow Impose Function]

This would be quite an unusual type of imposed function; more usually these function will be connected by a state vector. For example, if we wish to periodically check the number of attendance of a student or the last time they attended, we would simply need to inspect the local variables of the student process, also known as the state vector. Hence:

![Diagram of Snapshot Impose Function]

Finally, the most usual type would require the inspection of many model processes e.g. all students. So if the information request was list all students whose current attendance is less than 75%, we would have the SSD:
5.2.4. Interactive Functions

An interactive function is, as the name implies a function that interacts with the system, it takes information from the system, processes it, and outputs it back into the system. For example, a file backup function creates output (the backup file) which is used as input to a restore function.

To continue our student attendance theme, if the model of the student process included a number of ways in which a student ceases to be so (graduates, drops out, kicked out through poor attendance) and we wish to automate this event as much as possible, we might extend the system specification such that the poor attendance function produces a data stream containing the details of students who qualify for being removed which is then fed back into the student model process and rough merged with the original model input. Hence:

5.2.5. Filter Functions

Input entry problems and data validation should be kept separate from the model processes; as far as possible, only valid data should be passed to the model processes. Hence in JSD, we can validate input data by the use of filter functions to detect and if possible correct any data that is invalid in the sense of having incorrect values or in a sequence that the model process is not designed to recognised.

Invalid data are detected and dealt with in appropriate filter processes that can be designed by producing process structure diagrams using the POSIT/ADMIT construct. For example, valid data are modelled as a POSIT and QUITS are inserted into this structure to deal with validation errors. All QUITS lead to the ADMIT structure where intolerable side-effects must be dealt with (see JSP notes).

When the filter function deals with the sequence of arrival of input messages, then we are producing a context filter function. Again POSIT/ADMIT structures may be useful here, because again we are dealing with uncertainty. Note that detection is a lot easier than recovery, because there are a large number of possible recovery options. For example, given a student model process which has a sequence of two events: enrol followed by an iteration of attend lectures events,
we have a structure text:

Student Seq
  Enrol;
  Attendance Iter (while still a student)
  Attend Lecture;
  Attendance End
Student End

Hence a context filter must ensure that an enrolment input message is the first one and can only
be followed by attend lecture messages. consider the following context filter expressed In struc-
tured text; which one is right?

Student Seq
  Read (message);
  Enrol Sel (if an enrol message);
    Write (message);
    Read (message);
  Enrol Alt
    Write ('error message');
    Generate dummy enrolment;
    Write (dummy);
  Enrol End
  Attendance Iter (while still a student)
    Attend Lecture Iter (while attend message);
      Write (message);
      Read (message);
    Attend Lecture End
  Spurious Message Iter
    Write ('ignored message');
    Read (message);
  Spurious Message End
  Attendance End
Student End

Student Seq
  Read (message);
  Remove Invalid Message 1 Iter (while not enrol)
    Write ('error message');
    Read (message);
  Remove Invalid Message 1 End
Enrol Seq
  Write (message);
  Read (message);
  Remove Invalid Message 2 Iter (while not attend)
    Write ('error message');
    Read (message);
  Remove Invalid Message 2 End
Enrol End
  Attendance Iter (while still a student)
    Attend Lecture Seq
      Write (message);
      Read (message);
    Remove Invalid Message 3 Iter (while not attend)
      Write ('error message');
      Read (message);
    Remove Invalid Message 3 End
  Attend Lecture End
  Attendance End
Student End

5.2.6. A Word of Warning

The specification of functions allows the systems designer a large amount of freedom and inevita-
ably exercises his inventive powers and ability to choose from a range of alternatives. However,
such functions are always specified as processes with long lifetimes (the same as model processes). For example, we do not specify a weekly sales report, we specify instead the sequential process whose output is the set of all weekly sales reports. This is because, at this stage at least, specifying such processes gives us a better opportunity of identifying the relationships with the system model and the whole lifetime of the process.

5.3. Adding Time to the Model

In our model and functional processes the passage of time is marked only by the occurrence of actions, e.g. a library member returns a book some time after they have loaned it. To achieve specific timing between actions we need to introduce special messages called *time grain markers* (TGMs). TGMs are sent to processes as a separate specialised data stream and are read as input records. The processes look for their arrival before a special action is allowed to happen. A frequent use of TGMs is the control of rough merged data streams. As discussed earlier, rough merge data streams exhibit no favouritism to any of the constituent data streams, and if one is written faster than another it will tend to get more attention. Sometimes this has to be controlled and TGMs are one way of doing it.

Therefore, a process which needs to be affected by real world time will have a Read(TGM) operation which will cause the process to be blocked until the arrival of the TGM.

For example, consider the simple process for an employee clocking into work:

```plaintext
Employee Init (while still employed)
  Day Seq
  Read (TGM);
  8.30am TGM
  Possible clock in Iter (while still time & not clocked in)
    Possible record Sel (clock data stream not empty)
      Read (clock data stream);
      Clock in;
    Possible record Alt (TGM data stream not empty)
      Read (TGM);
      9.00am TGM
    Possible record End
    Possible clock in End
    Day End

Employee End
```

N.B. The above makes no allowances for error messages.

When it is necessary to have a number of TGMs, it may be necessary to create a process which is responsible or the production and synchronisation of the TGMs. For example:
Note the special notation for the clock process.

5.4. Adding Functions to Our Case Study

5.4.1. Information Functions

Let us suppose that our friendly user has asked for an exception report for any order which is amended more than twice in a week. First we examine the process ORDER to see if we can embed write operations in it. Here is the structure text for ORDER:

```
Order Seq
  Place;
  Delays & Amends Iter (while order not dealt with)
    Action Sel (if an amendment)
      Amend:
      Action Alt (if a delay)
    Delay:
    Action End
  Delays & Amends End
  Deliver or cancel Sel (if completed)
    Complete Seq
      Allocate;
      Deliver;
      Complete End
  Deliver or cancel Alt (if cancelled)
    Cancel;
    Deliver or cancel End
Order End
```

- Can we embed writes successfully?
- Where would we allocate write (exception) operations?

The required structure for producing the exception report is quite unlike the structure of ORDER.

Attempt to produce a process structure for the function.
This functional process is connected to the ORDER process in the SSD by a data stream connection because the initiative for the actions we are concerned with (Amend and Delay) are contained within the ORDER process. We also need to introduce a TGM data stream which is rough merged with this new data stream. This gives us the revised SSD:

Now write the structure text for our new process with the necessary read operations and operations necessary to count the occurrences of the amendments and output the exception report as a single line. Do not show the rough merge reads explicitly but as the simple operation Read(EXC+TGM WEEK).

```plaintext
Exception Function Seq
   Read (EXC-TGM WEEK);
   Exception function body Iter
      Week Seq
         count := 0;
         Amend Group Iter (while amend and count < 2)
            count := count + 1;
            Read (EXC-TGM WEEK);
          Amend Group End
         Extra Amends Iter (while amend)
            write exception report line;
            Read (EXC-TGM WEEK);
          Extra Amends End
      Read (EXC-TGM WEEK);
   Week End
   Exception function body End
Exception Function End
```
Remembering the basic operations for a rough merge read (if data stream 1 not empty read(data_stream 1) else if data stream 2 not empty read(data_stream 2)), work through the above to ensure that the required weekly report would be produced.

- What amendments are required to the ORDER process?

5.4.2. Major Functions

We decide that the clerk's functions allocate and delay would be automated by system functions. We now need to firm up on the precise meaning of these activities. For example, when is allocation to be done? how are delays minimised? Discussion with the user should elicit the precise nature of these activities and their relationship with ORDER and CUSTOMER processes.

Let us assume that the allocation policy is as follows:

- Allocation is always for a specific product, there are no allowable alternatives;
- If an order is delayed then it receives a higher priority in the next allocation;
- Allocation of stock is done on a daily basis after the stock update is completed.

This means we must amend our SSD to include a new function process ALLOCATE FUNCTION for each product. It is initiated by a TGM data stream and is connected to the PRODUCT process by means of a state vector connection (in order to obtain availability information). It also needs to interact with orders, hence we can write a data stream from the ORDER process with details of all orders received and introduce a further process ORDER RECEIVED which reads this data stream and is also connected to ALLOCATE FUNCTION by three data stream connections: one to receive quanreys of quantity required (rough merged with all orders received); one as a reply to the enquiries; and one to send, allocate or delay messages. Try producing the revised SSD.
6. JSD Implementation Stage

6.1. Implementation Strategy

A JSD SSD describes a number of concurrently running processes. Theoretically, each of the processes could run on individual processors giving a true concurrency. Obviously, this will be uneconomical for most systems, so we must schedule the processes to run as required, normally sharing one or perhaps two processors.

Usually this means allowing all the processes to execute on one processor. In other words we must schedule the time available for a given process to be allowed to use the processor, and we must organise a schedule that dictates the order by which the processes have access to the processor.

6.1.1. How Many Processors are to be Used?

The first step in deciding the implementation strategy is to decide how many processors are to be used. For example, are we to have a centralised or a distributed system? We might decide to partition the SSD into implementation units each with its own processor. For example, we might decide in conjunction with our users that the basic data capture will be done on a local basis using local microcomputer, but the major data handling processes will be done centrally on a mainframe.

6.1.2. Real-Time or Batch?

Next we would want to choose the type of implementation; for example, batch processing or real-time. We would also determine the extent to which the operating system(s) available can help in scheduling the processes, perhaps on a time-slice basis.

6.1.3. Will a Scheduler be Required?

Because we are in general constrained by von Neumann sequential processors we will need to introduce a scheduler process to control the system's activity. Jackson states that we can get away without having a scheduler when there are only data stream connection, and there are no rough merges, and when any two processes are connected by only one path.

6.2. System Implementation Diagram

SIDs show the calling sequence of processes as possible lines of communication. (note the conditions for calling a process are only shown in the program structure diagrams or in the structured text or the scheduler process).
The notation for the basic components of a SID are as follows:

Here we have two processes, where the process PB is *inverted* with respect to one of its data streams and is called by the process PA. (The parallel lines indicate one communication channel between the processes).

Here we have two processes, where the process PY is *inverted* with respect to three of its data streams and is called by the process PX. (The parallel lines indicate three communication channel between the processes).

Here we show the special-purpose scheduling process, the *Scheduler*. This is introduced at the implementation stage and hence is not found in the SSD. Scheduling the processing order of the processes and functions is not the only task that the scheduler performs, its purpose is to manage the following tasks:

1. to handle system output;
2. to evaluate input and call correct processes for that input;
3. to monitor the status of processes and if necessary terminate one and start another;
4. to maintain buffers for any data passed between processes;
5. to monitor and maintain state vector files to determine order of process execution.
This is a buffer. These are used to store waiting messages when processes are connected by process inversion and the data stream messages are not processed on a one-for-one basis.

This is used to show a direct access file of state vectors of process PQ.

Here, we show the two dismembered parts of the process P; executed by the scheduler process. Dismembered part P-a reads a part of the data stream D, as does dismembered part P-b.

6.3. Scheduler Processes

In general, a scheduler process will be responsible for accepting system input (normally via a filter process) and scheduling the system processes. It may also be responsible for buffer and state vector file management.

A scheduler process is shown on the SID as above, but must be described in detail by a program structure diagram and possibly structure text.

A typical scheduler process may be represented as a menu selection type process. For example, a PSD such as:
This might give rise to the following structure test:

```
Scheduler Seq
  Read (message);
  Scheduler body Iter (while not EXIT)
    Message Sel (DEBIT)
      Debit action;
      Call Debit-process (amount);
    Message Alt (CREDIT)
      Credit action;
      Call Credit-process (amount);
    Message Alt (ENQUIRY)
      Enquiry action;
      Loadsv (account);
      Call Enquiry-process (amount);
  Message End
  Read (message);
Scheduler body End
Scheduler End
```

Notice the Call and the Loadsv (access state vector file) operations. There might also have been a Storesv (update state vector file) and read and write buffer operations.

### 6.4. Process Inversion

The purpose of process inversion is to make a given process a subprogram of the scheduler, another process or processes, or both. Process inversion converts the concurrent model into a sequential model (refer to JSP notes). It allows two or more sequential processes to be easily and conveniently scheduled on a single processor.

Inversion produces a hierarchy of a main program and subprograms with communication via parameters; hence inverted processes run as required at the dictate of the main program.

Inversion changes the link between two processes from a file or buffer based connection into a direct call in which any data are passed as parameters. Hence we say that a process is inverted with respect to its former connection to other processes. Note that two inversions are always possible. When process A is connected to process B by data stream DS, we can invert A to B in respect of DS, or B to A (i.e. either process can be the subprogram).

We start with a network of freely running concurrent programs exchanging data with each other and the real world via data streams and state vector data connectors. The diagram below shows a very simple SSD:
We cannot schedule the whole program at once, we must follow a logical path and only deal with two processes at any one time. The inversion is carried out along a data connection; we only invert processes that communicate directly with one another. Furthermore, we only need to consider processes that communicate via data stream connections; we will deal with state vectors later.

The following procedure is used:

Select two processes that communicate via a data stream. If the two processes are freely running on their own processors, then the writer can transmit records one after the other forever; the data stream buffer will store them until the reader wishes to read them. Inversion solves two problems; firstly by inverting one with respect to the other only one processor is required; secondly the unbounded buffer is no-longer required as the writer will only get the chance to write one record before it must halt until the reader process reads it.

Therefore, the relationship between the two processes can take one of two forms:

1. The writer process runs until one record of data is produced and is ready for transmission. The writer process halts calling the reader process. The reader process starts and runs until it has processed the record of data and requires another. The reader process halts and returns control to the writer process which runs until it has produced another record of data and so on. This is known as Supply Driven Inversion.

2. The reader process runs until it requires a record of data. The reader process halts by calling the writer process. The writer process runs until it produces a record of data. The writer process then halts and returns control to the reader process which runs until it requires another record of data and so on. This is known as Demand Driven Inversion.

In implementation terms:

- In the calling process, replace the Write (data stream message) instructions
with Call Subprogram (data stream message);

- In the called process (subprogram), replace the Read (data stream message) with Exit from Subprogram.

Process text pointers are required to implement most JSD processes. Recall that these are simply pointers which indicate which section of code will be executed each time the process is called according to the input message available to it. This is a similar concept to the implementation of JSP inversion where a state variable was necessary in order to achieve re-entry to a subprogram at the correct place (i.e. at the place of last exit). This also means that the code must be re-entrant.

6.5. Transforming from System Specification to System Implementation

We start by identifying a scheduler, this can either be a new process or an existing one adapted. Generally the correct procedure after the identification of the scheduler is as follows:

1. Start with the input processing processes, the system must have input first, therefore input processes must be dealt with first.

2. Invert all input processes with respect to the scheduler, i.e. they are placed below the scheduler on the SID.

3. Follow the sequence of data stream connectors in the SSD and continue inverting the next neighbour process in time below the last. Processes follow each other and should call each other passing data along as it is processed.

4. Look for functional processes connected only by state vectors. These are likely to be reports and query screens and will probably be inverted with respect to the scheduler separately so they can be called by user demand (which remember is via a connection to the outside world and therefore is a data stream data flow which gives us the inversion path).

5. Examine user interface requirements to see if any processing sequences are required as separate options and therefore have to be called separately.

In our example, a scheduler can be added with the two input data streams transferred to it, we can therefore invert the process Process Order and the function Monitor Orders with respect to the scheduler. We then look at those processes connected to Process Order. Two processes are connected to it via data streams. Next, therefore, we can invert Process Goods with respect to Process Order and thus remove the data stream Goods Detail. We now have an example of a multi-level inversion, a typical feature in which an inverted process itself calls another process. The procedure can be further simplified by grouping the inverted processes together and showing them as a single inverted program with respect to the scheduler. It is not necessary to do this but with larger diagrams it often helps to make the model as simple as possible.

6.5.1. Resolution of Data Streams

So far we have been happily disposing of data streams. However, what happens when there are more than one type of record flowing down a data stream? In a fixed merge data stream relationship there are only two processes involved, therefore we can remove the virtual need for a buffer and implement the data stream as a process inversion relationship. However, this is not the case with rough merged data streams.

In our example, Process Goods and Process Invoice are both connected to Produce Report by data streams that are rough merged. So which process calls which and which process passes data and when? We can break the problem into two parts: the inversion and the problem of what happens to the data if there is more than one type of record present.
The inversion problem can be solved relatively easily. Looking at the problem from the reader's viewpoint, the problem is because the data arrives from any of the writers of the rough merge and we do not know which process to run to obtain data; if we run one are we being unfair to the others? However, if we look at the problem from the writer's viewpoint then life is easier. The writer is unaware of the rough merge as far as it is concerned it is writing records of data to the reader. Therefore when it has produced a record of data it will want to call the reader to process it. What happens then is that whenever one of the writers involved in the rough merge is running and produces the record of data, it calls the reader process to process it. When the reader has processed the data it passes control back to the writer. *Supply Driven Inversion* is used.

6.5.2. Further Example 1

The SSD:

The diagram shows the SSD with nodes labeled A to G and arrows indicating the flow of data. The nodes are connected in a specific order, suggesting the sequence in which data is processed.
Yields the following SID using a scheduler:

```
A
D

W
X

Y
Z

C
G
```

Or, in this case we can also invert without using a scheduler:

```
X

A
W
D

Y

G
Z
```

Here, process X takes on the scheduling activities. By effecting different inversions, can you produce a different SID without using a separate scheduler? What will be the effect on the scheduling algorithms?
6.5.3. Further Example 2

The SSD:

![Diagram of SSD]

Gives a SID using a scheduler:

![Diagram of SID]

6.5.4. State Vector Separation

We now need to consider the state vectors. Remember that in a state vector connection there is no concept of the writer actively writing to the inspector process, the inspector process merely inspects the writer's state whenever it wishes. In a scheduled system it is likely to be impossible for the inspector and the writer to be running at the same time, when the inspector needs to inspect the state vector of the writer it has a problem because the writer is suspended and removed from the processor and its memory.

The solution is straightforward, the state vectors are separated out from their parent processes. In other words, instead of the state vector just being the variables in memory as the process runs, when the process halts the contents of the processes state vector are dumped into a file. So that when a running process wishes to inspect the current state vector of a process it returns control to the scheduler which then obtains the necessary record of information from the state vector file and passes it back to the inspector as a parameter when the inspector resumes. The information still reflects the current state of the writer because the state vector file is updated every time the writer process halts.

The process that requires the state vector information is usually connected to the scheduler by a multi-channelled connector to reflect the various state vector
records being passes through the calling mechanism. Each occurrence of the process will have its own state vector associated with it—each customer obviously has their own identification, balance etc.

It should be noted that the identification of marsupial entities in effect normalises the data to 1st normal form (by removing repeating groups). Jackson claims that further normalisation is unnecessary because the modelling process concentrates on actions and defines the relationships between data. (see Sutcliffe section 5.7 for a more detailed discussion of the relationship of JSD to data analysis)

6.5.5. Implementing the Case Study

We will now refer to the SSD given at the end of chapter 6, and using the guidelines above, produce an SID. We will simplify the SSD by replacing the Enquiry and Reply data stream connections by the state vector connection Order SV. The data stream All-Orders will be rough merged with Allocation.

We will make the assumption that there is to be one processor and that we will have a scheduler which receives a combined system input file from a filter process which has rough merged the system input.

An over-simplified assumption may be made that we are not at this stage concerned with security and backup procedures. Also, we will not concern ourselves with the omission of order listing functions (although they would be a trivial exercise to include on the SSD and hence the SID).
We will need to decide on the state vector files necessary and we can reasonably decide to combine the state vectors of the order process and the order receive process.

Now, we need to complete the implementation details by ensuring we have complete and up to date PSDs and if desired structure texts for each process so that the necessary inversions and file accesses can be demonstrated. For our purposes let us first add appropriate Read (data stream) and Getsv (state vector) and any other appropriate embedded functions to the structure text given below. We have omitted the filter process to save time. The scheduler process will be dealt with later.

<table>
<thead>
<tr>
<th>Customer Iter (while still a customer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Sel (if a new order is placed)</td>
</tr>
<tr>
<td>Action Alt (if an order is amended)</td>
</tr>
<tr>
<td>Action Alt (if an order is delivered)</td>
</tr>
<tr>
<td>Action Alt (if an order is cancelled)</td>
</tr>
<tr>
<td>Action End</td>
</tr>
</tbody>
</table>

Customer End
<table>
<thead>
<tr>
<th>Product Seq</th>
<th>Product body Iter</th>
<th>Product body End</th>
<th>Product End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Available;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unavailable;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order Seq</th>
<th>Order received Seq</th>
<th>Allocate Iter</th>
<th>Allocate delayed Iter (while delayed orders)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>requested &lt;= available)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>requested &gt; available)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allocate delayed End</td>
<td>Allocate normal Iter (while delayed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>requested &lt;= available)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>requested &gt; available)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allocate normal End</td>
<td>Allocate End</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Place:**

- Amend Iter (while amendments)

<table>
<thead>
<tr>
<th>Amend End</th>
<th>Deliver or cancel Sel (if cancel)</th>
<th>Deliver or cancel Alt (if deliver)</th>
<th>Deliver or cancel End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Amend;</td>
<td>Cancel;</td>
<td>Deliver;</td>
</tr>
</tbody>
</table>

**Allocate Iter**

- Allocate delayed Iter (while delayed orders)

<table>
<thead>
<tr>
<th>Allocate delayed End</th>
<th>Allocate normal Iter (while delayed)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Allocate normal End</td>
</tr>
</tbody>
</table>

- Allocate delayed Sel (if requested <= available)

<table>
<thead>
<tr>
<th>Allocate;</th>
</tr>
</thead>
</table>

- Allocate delayed Alt (if requested > available)

<table>
<thead>
<tr>
<th>Delay;</th>
</tr>
</thead>
</table>

- Allocate normal Sel (if requested <= available)

<table>
<thead>
<tr>
<th>Allocate;</th>
</tr>
</thead>
</table>

- Allocate normal Alt (if requested > available)

<table>
<thead>
<tr>
<th>Delay;</th>
</tr>
</thead>
</table>

- Allocate normal End

<table>
<thead>
<tr>
<th>Normal order End</th>
</tr>
</thead>
</table>
### Chapter 6 - JSD Implementation Stage

<table>
<thead>
<tr>
<th>Customer Iter (while still a customer)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Cust Input;</td>
<td></td>
</tr>
<tr>
<td>Action Sel (if a new order is placed)</td>
<td>Place;</td>
</tr>
<tr>
<td></td>
<td>Write</td>
</tr>
<tr>
<td>Cust Order (new order);</td>
<td></td>
</tr>
<tr>
<td>Read Cust Input;</td>
<td></td>
</tr>
<tr>
<td>Action Alt (if an order is amended)</td>
<td>Amend;</td>
</tr>
<tr>
<td></td>
<td>Write</td>
</tr>
<tr>
<td>Cust Order (amendment);</td>
<td></td>
</tr>
<tr>
<td>Read Cust Input;</td>
<td></td>
</tr>
<tr>
<td>Action Alt (if an order is delivered)</td>
<td>Deliver;</td>
</tr>
<tr>
<td></td>
<td>Write</td>
</tr>
<tr>
<td>Cust Order (delivery);</td>
<td></td>
</tr>
<tr>
<td>Read Cust Input;</td>
<td></td>
</tr>
<tr>
<td>Action Alt (if an order is cancelled)</td>
<td>Cancel;</td>
</tr>
<tr>
<td></td>
<td>Write</td>
</tr>
<tr>
<td>Cust Order (cancellation);</td>
<td></td>
</tr>
<tr>
<td>Read Cust Input;</td>
<td></td>
</tr>
<tr>
<td>Action End</td>
<td></td>
</tr>
<tr>
<td>Customer End</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product Seq</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Getsv Product sv</td>
<td></td>
</tr>
<tr>
<td>Product body Iter</td>
<td></td>
</tr>
<tr>
<td>Unavailable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Getsv Product sv</td>
<td></td>
</tr>
<tr>
<td>Product body End</td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Product End</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order Seq</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Cust Order;</td>
<td></td>
</tr>
<tr>
<td>Place;</td>
<td></td>
</tr>
<tr>
<td>Read Cust Order;</td>
<td></td>
</tr>
<tr>
<td>Amend Iter (while amendments)</td>
<td>Amend;</td>
</tr>
<tr>
<td></td>
<td>Write</td>
</tr>
<tr>
<td>Exception;</td>
<td></td>
</tr>
<tr>
<td>Read Cust Order;</td>
<td></td>
</tr>
<tr>
<td>Amend End</td>
<td></td>
</tr>
<tr>
<td>Deliver or cancel Sel (if cancel)</td>
<td>Cancel;</td>
</tr>
<tr>
<td>Read Cust Order;</td>
<td></td>
</tr>
<tr>
<td>Deliver or cancel Alt (if deliver)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deliver;</td>
</tr>
<tr>
<td>Read Cust Order;</td>
<td></td>
</tr>
<tr>
<td>Deliver or cancel End</td>
<td></td>
</tr>
<tr>
<td>Order End</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order received Seq</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Allocation &amp; All Orders;</td>
<td></td>
</tr>
<tr>
<td>Allocate or delay Sel (if sufficient stock)</td>
<td>Allocate;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocate or delay Alt (if insufficient stock)</td>
<td>Delay;</td>
</tr>
<tr>
<td>Allocate or delay End</td>
<td></td>
</tr>
<tr>
<td>Order received End</td>
<td></td>
</tr>
</tbody>
</table>
Allocate Iter
  Read TGM Allocate;
  Getsv Product Available;
  stock_available := PAv stock;
  Getsv Order Vector;
  Allocate delayed Iter (while delayed orders)
    Delayed order Sel (if requested <= available)
      Allocate;
      stock_available := requested;
      Write Allocation (allocate);
    Delayed order Alt (if requested > available)
      Delay;
      Write Allocation (delay);
  Delayed order End
  Getsv Order Vector;
  Allocate delayed End
Allocate normal Iter (while delayed)
  Normal order Sel (if requested <= available)
    Allocate;
    stock_available := requested;
    Write Allocation (allocate);
  Normal order Alt (if requested > available)
    Delay;
    Write Allocation (delay);
  Normal order End
  Getsv Order Vector;
  Allocate normal End
Allocate End

Exception Function Seq
  Read (Exception + TGM Week);
  Exception Function body Iter
    Week Seq
      count := 0
      Amend Group Iter (while amend and count < 2)
        count := count + 1;
        Read (Exception + TGM Week);
      Amend Group End
    Extra amends Iter (while amend)
      Write Exception Report Line;
      Read (Exception + TGM Week);
    Extra amends End
    Week TGM Seq
      Read (Exception + TGM Week);
    Week TGM End
  Exception Function body End
Exception Function End

Now, after appropriate checking, we can add the instructions necessary for the indicated inversions. We will assume this is to be done and next produce a PSD and structure text for the scheduler. Recall, all system inputs arrive validated; the scheduler must invoke the customer process when customer input is read and allocate the exception functions when signalled by the TGM input.
The next step would be state vector separation to produce the appropriate state vector files. Let us look at Customer SV File. First we must examine he process and any state vector connections in order to elicit the entity attributes. These are not the same as action attributes which are input data messages consumed by actions. Entity attributes are created and updated by an entity’s actions and as a result record the entity’s life history.

The first and most obvious is the CUSTOMER-ID as a key. The second is the TEXT-POINTER to indicate where the customer model is in respect to its life history. surprisingly, others do not come readily to mind. This is because our modelling has not revealed a need for any information concerning the status of a customer. This might change if course, for example, if the users wish to know how many orders a customer has made. A more productive exercise might be to look at the entity attributes of the order process and the order received process. (Note that the allocate function is connected via a state vector connection to the order received process because it requires knowledge of what product has been ordered and the quantity).

### 6.6. Coding

The transition to coding involves the addition of detail to the PSDs and/or structure text. for example, physical design detail such as initialisation of files or variables.

This step is taken more or less from JSP (see JSP notes) in that we list operations (i.e. the detail of actions or computer-related operations) and conditions; then assign them to the PSDs (struc-
ture text). These operations and conditions are refined together with the control structures inherent in the PSD to produce target language code.

Backtracking (in filter processes for example) is implemented using the POSIT/ADMIT and QUIT structures of JSP.

Inversion again is implemented using the techniques already described in JSP. In JSD the text pointer of the process state vector is used to control the flow of an inverted subprogram. (Recall control passing mechanisms in JSP implementations).