Notes

6. Network Stage Elaboration Phase

6.1. 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.

6.1.1. 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.

6.1.2. 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.

6.2. Adding Functional Processes

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

6.2.1. 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 <u>not</u> 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

Notes

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.

6.2.2. 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.

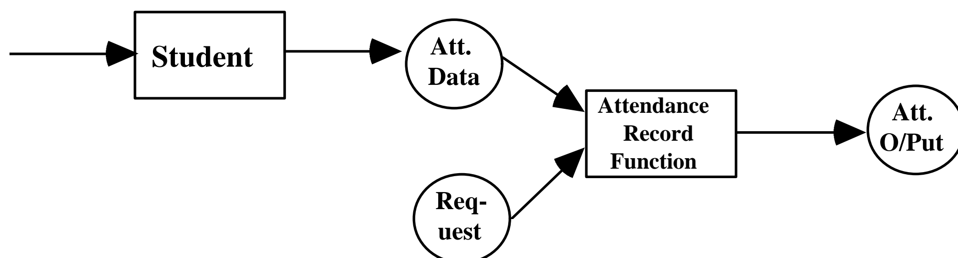
Imposed functions are frequently used to inspect the state vector of their connected process(es) and are therefore usually of report writing or query screen nature. Since most state vectors are, in implementation terms, files, many data base query programs are in effect *imposed* functions.

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 fulfil the system specification. The requirements for them also tend to be rather fluid a high maintenance area.

6.2.3. 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.

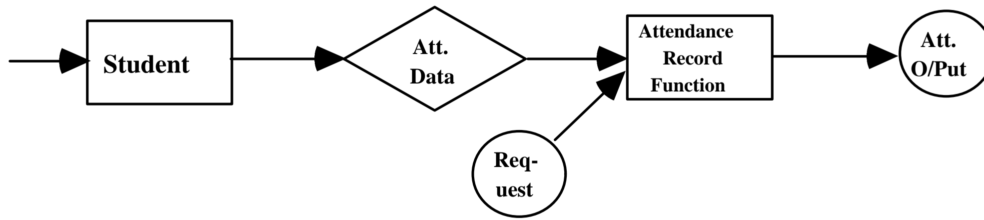
If we wished to examine periodically the pattern of attendance of a student (i.e. look at his attendance record over the period since we last looked), we might have a model process which outputs a data stream containing the attendance details which is rough merged with a user request data stream as input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



Example of Continuous Flow Impose Function

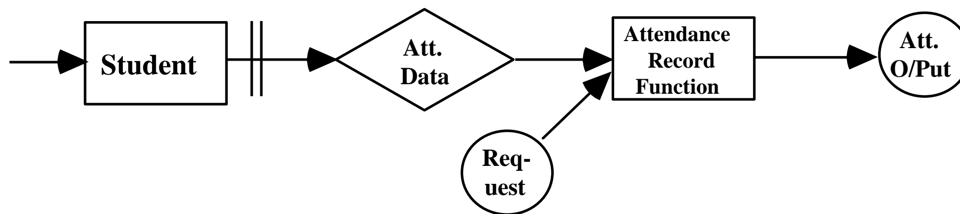
This would be quite an unusual type of imposed function; more usually these function will be connected by a state vector. For example, if we wish to periodically check the number of attendance s of a student or the last time they attended, we would simply need to inspect the local variables of the student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

Notes



Example Snapshot Imposed Function

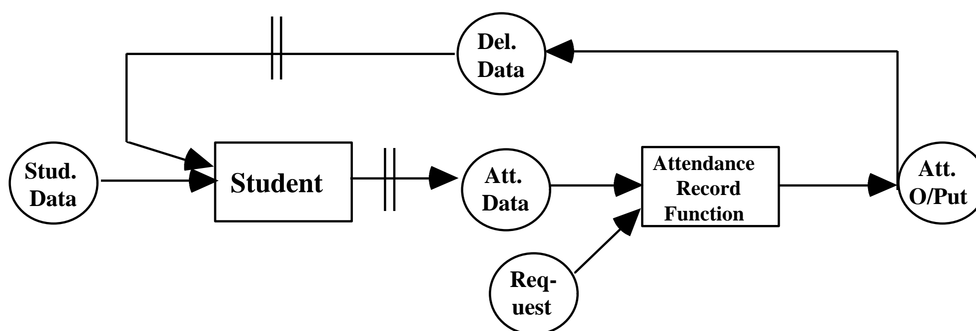
Finally, the most usual type would require the inspection of many model processes e.g. all students. so if the information request was list all students whose current attendance is less than 75%, we would have the SSD:



6.2.4. 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 crease 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:



6.2.5. 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.

Notes

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 structured 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);
```

Notes

```
         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
```

6.2.6. A Word of Warning

The specification of functions allows the systems designer a large amount of freedom and inevitably 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). foe 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.

6.3. 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.

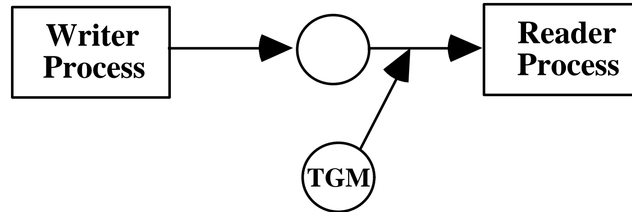
Therefore, a process which needs to be affected by real world time will have a Read(TGM) operation which will cause the process to be blocked until the arrival of the TGM.

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

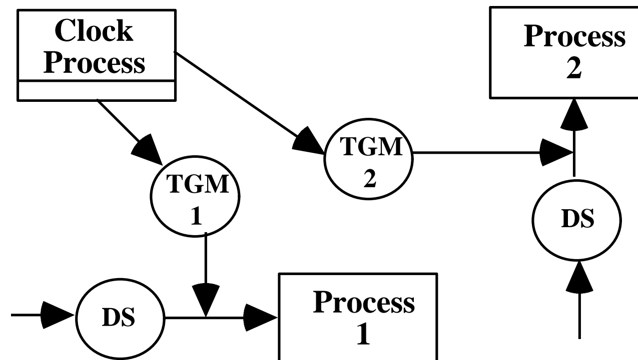
```
Employee Iter (while still employed)
   Day Seq
   Read (TGM);
   8.30am TGM;
   Possible clock in Iter (while still time & not clocked in)
      Possible record Sel (clock data stream not empty)
         Read (clock data stream);
         Clock in;
      Possible record Alt (TGM data stream not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

Notes



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.

6.4. Adding Functions to Our Case Study

6.4.1. 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
```

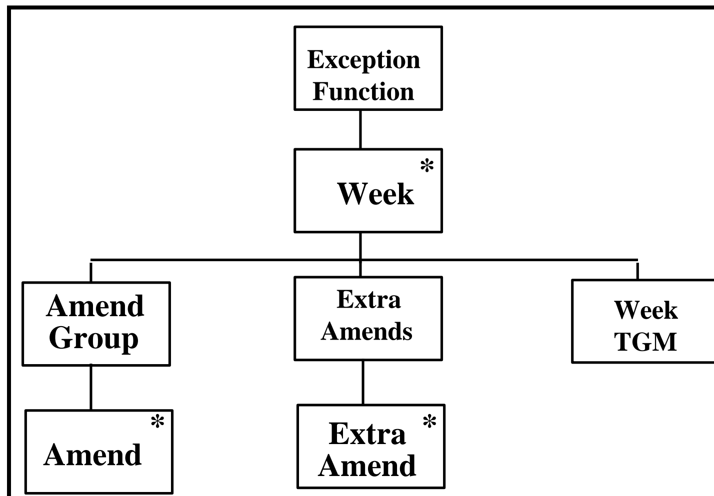
- Can we embed writes successfully?

Notes

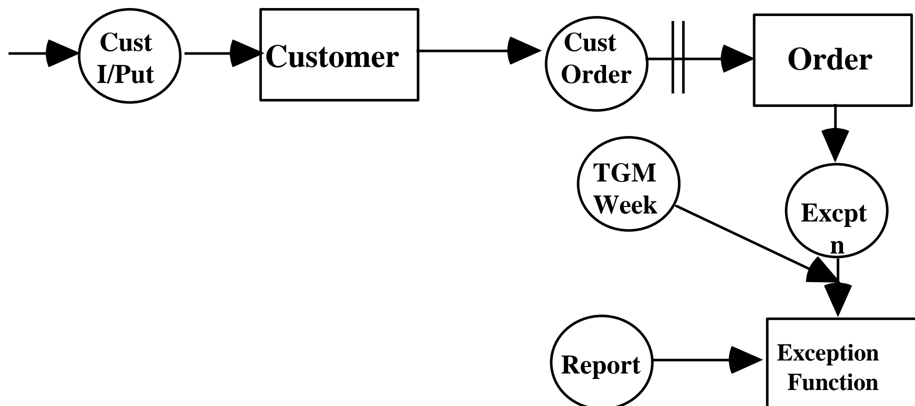
- Where would we allocate write (exception) operations?

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).*

```
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;
```

Notes

```
            Read (EXC+TGM WEEK);
         Extra Amends End
         Read (EXC+TGM WEEK);
      Week End
   Exception function body End
Exception Function End
```

Notes

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?

6.4.2. Major Functions

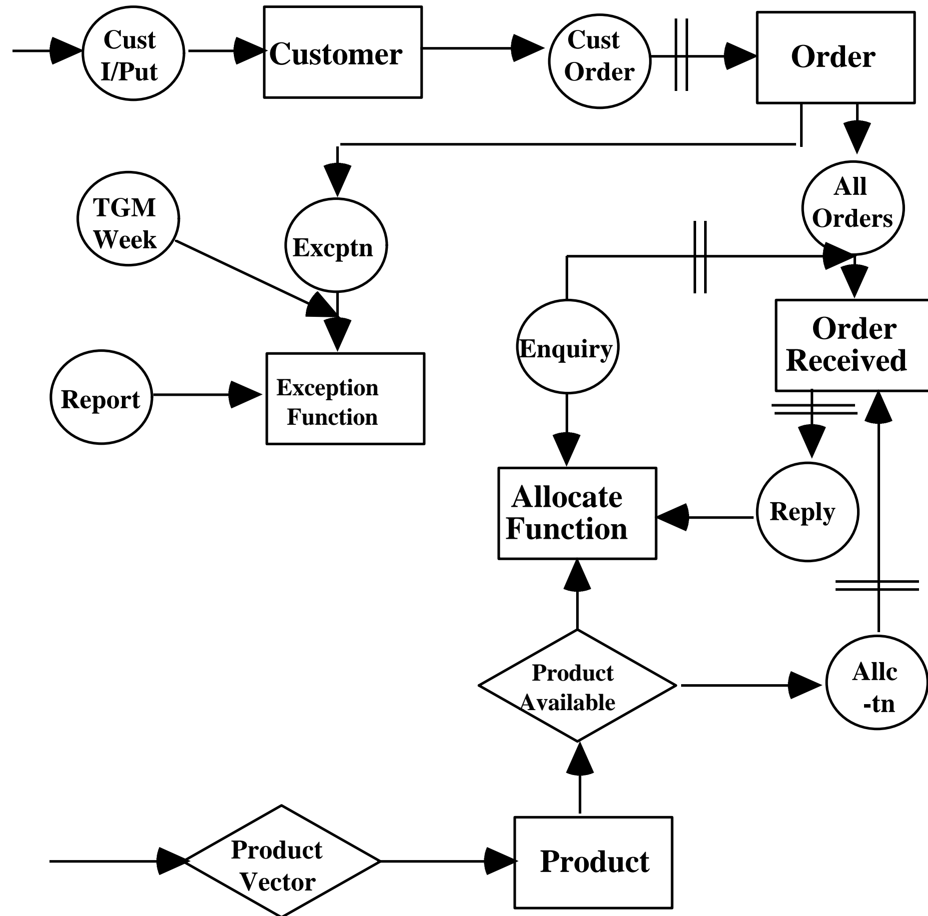
We decide that the clerk s functions *allocate* and *delay* would be automated by system functions. We now need to firm up on the precise meaning of these activities. For example, when is allocation to be done? how are delays minimised? Discussion with the user should elicit the precise nature of these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

Notes



Notes

7. JSD Implementation Stage

7.1. 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.

7.1.1. 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.

7.1.2. 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.

7.1.3. Will a Scheduler be Required?

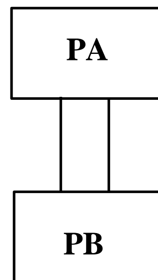
Because we are in general constrained by von Neumann sequential processors we will need to introduce a scheduler process to control the system s activity. Jackson states that we can get away without having a scheduler when there are only data stream connection, and there are no rough merges, and when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

7.2. System Implementation Diagram

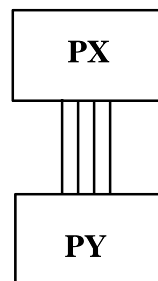
SIDs show the calling sequence of processes as possible lines of communication. (note the conditions for calling a process are only shown in the program structure diagrams or in the structured text or the scheduler process).

Notes

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).



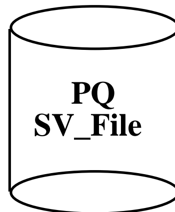
Here we have two processes, where the process PY is *inverted* with respect to three of its data streams and is called by the process PX. (The parallel lines indicate three communication channel between the processes).

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:

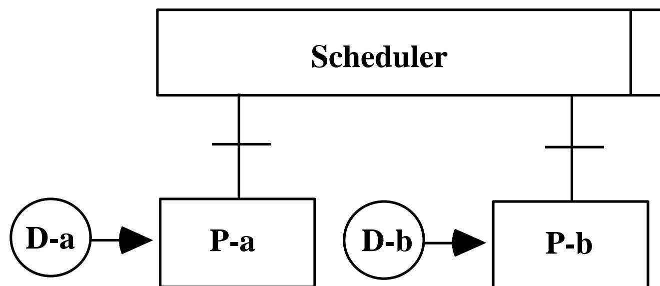
1. to handle system output;
2. to evaluate input and call correct processes for that input;
3. to monitor the status of processes and if necessary terminate one and start another;
4. to maintain buffers for any data passed between processes;
5. to monitor and maintain state vector files to determine order of process execution.

Notes

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.

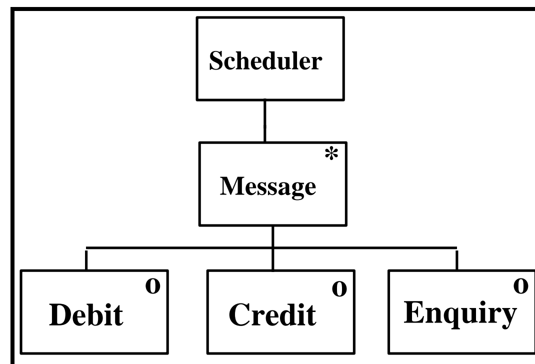
7.3. Scheduler Processes

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:

Notes



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.

7.4. Process Inversion

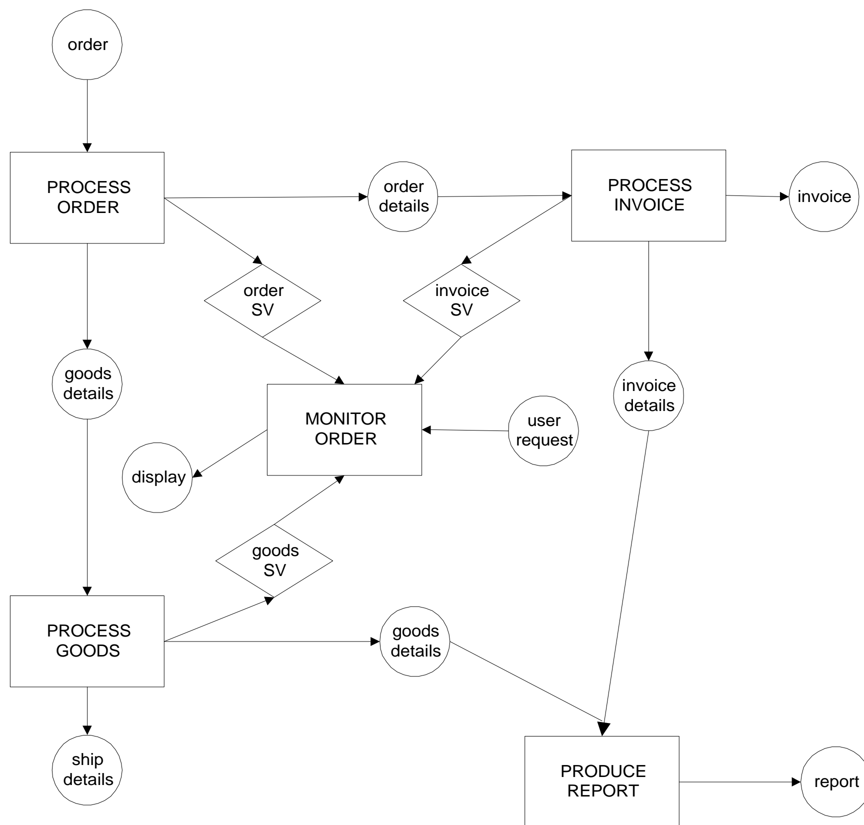
The purpose of process inversion is to make a given process a subprogram of the scheduler, another process or processes, or both. Process inversion converts the concurrent model into a sequential model (refer to JSP notes). It allows two or more sequential processes to be easily and conveniently scheduled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

Notes



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 one 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 processors 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 runes 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

Notes

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.

7.5. 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 place 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.

7.5.1. 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.

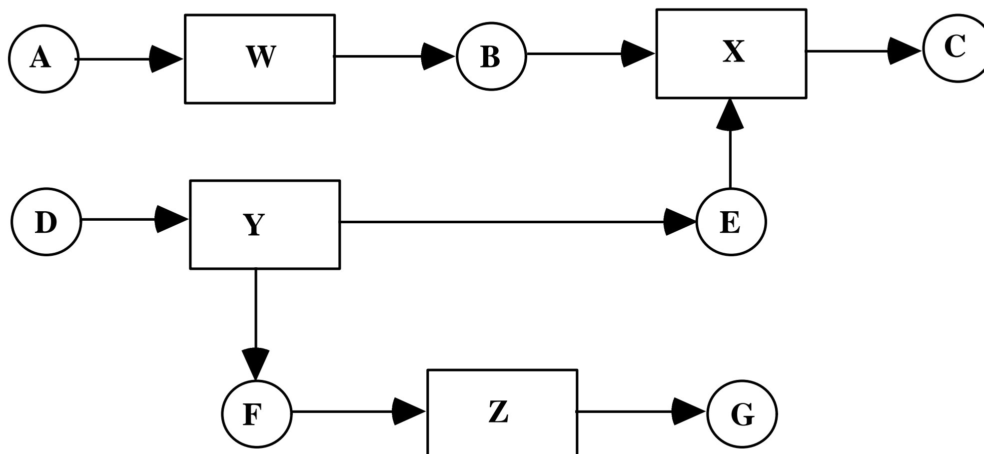
Notes

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 view point then life is easier. The writer is unaware of the rough merge ó 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.

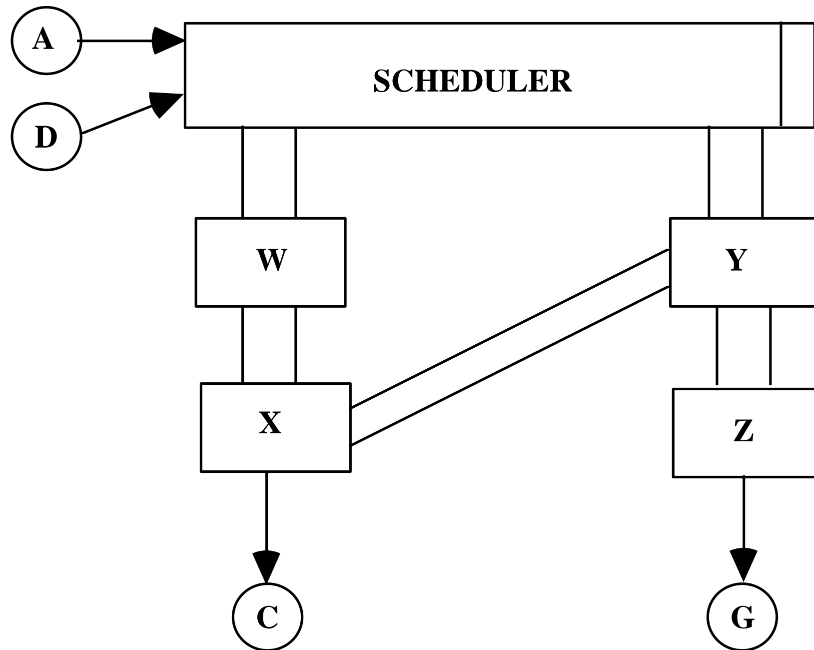
7.5.2. Further Example 1

The SSD:

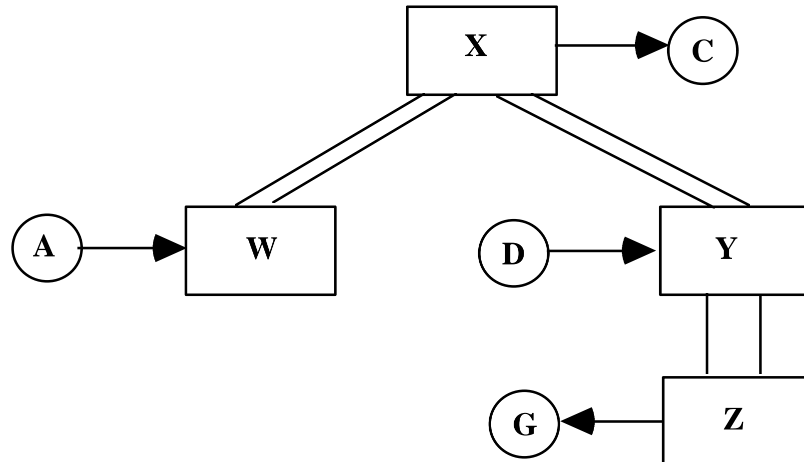


Notes

Yields the following SID using a scheduler:



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

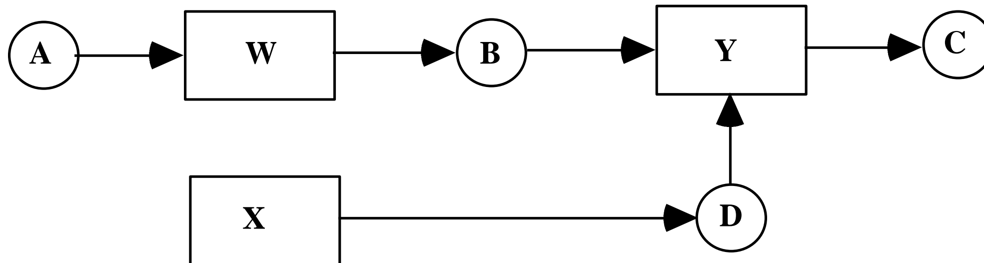


Here, process X takes on the scheduling activities. By effecting different inversions, can you produce a different SID without using a separate scheduler? What will be the effect on the scheduling algorithms?

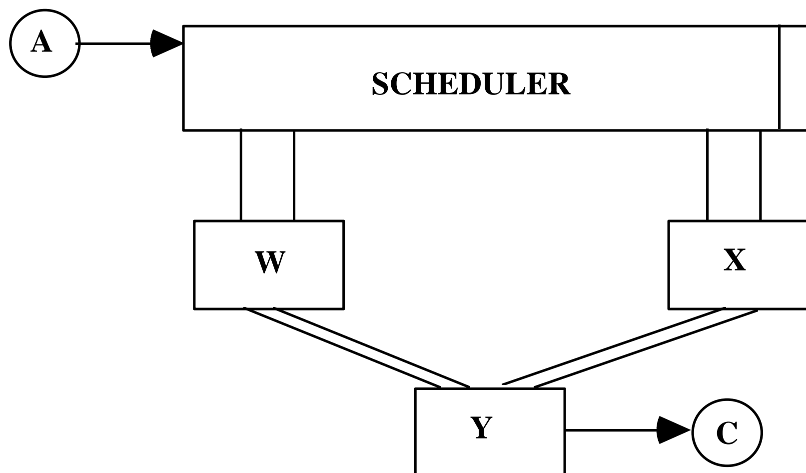
Notes

7.5.3. Further Example 2

The SSD:



Gives a SID using a scheduler:



7.5.4. 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 straight forward, 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

Notes

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)

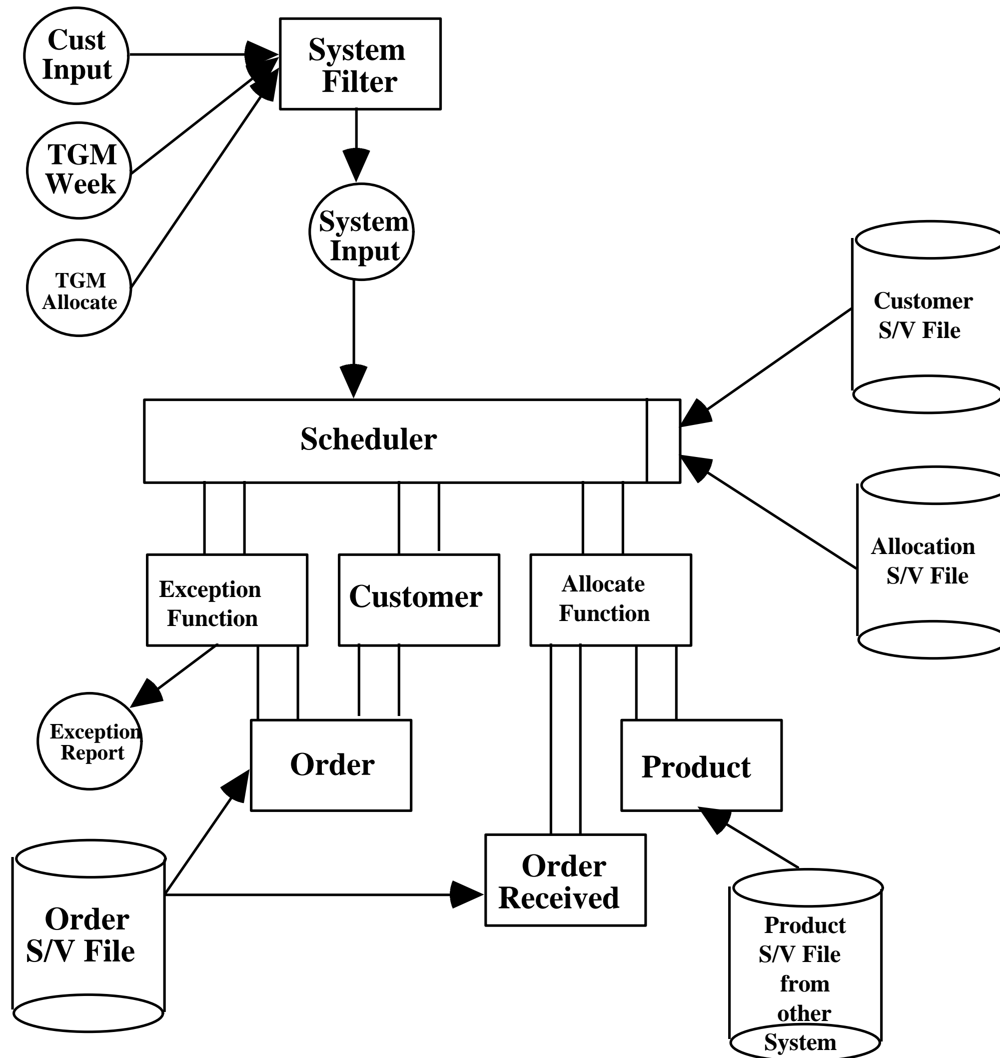
7.5.5. Implementing the Case Study

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).

Notes



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.

```
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
```

Notes

```
Product Seq
   Product body Iter
       Unavailable;
       Available;
   Product body End
Product End
```

```
Order Seq
   Place;
   Amend Iter (while amendments)
       Amend;
   Amend End
   Deliver or cancel Sel (if cancel)
       Cancel;
   Deliver or cancel Alt (if deliver)
       Deliver;
   Deliver or cancel End
Order End
```

```
Order received Seq
   Allocate or delay Sel (if sufficient stock)
       Allocate;
   Allocate or delay Alt (if insufficient stock)
       Delay;
   Allocate or delay End
Order received End
```

```
Allocate Iter
   Allocate delayed Iter (while delayed orders)
       Delayed order Sel (if requested <= available)
          Allocate;
       Delayed order Alt (if requested > available)
          Delay;
       Delayed order End
   Allocate delayed End
   Allocate normal Iter (while delayed)
       Normal order Sel (if requested <= available)
          Allocate;
       Normal order Alt (if requested > available)
          Delay;
       Normal order End
   Allocate normal End
Allocate End
```

Notes

```
Customer Iter (while still a customer)
   Read Cust Input;
   Action Sel (if a new order is placed)
       Place;
       Write Cust Order (new order);
   Read Cust Input;
   Action Alt (if an order is amended)
       Amend;
       Write Cust Order (amendment);
   Read Cust Input;
   Action Alt (if an order is delivered)
       Deliver;
       Write Cust Order (delivery);
   Read Cust Input;
   Action Alt (if an order is cancelled)
       Cancel;
       Write Cust Order (cancellation);
   Read Cust Input;
   Action End
Customer End
```

```
Product Seq
   Getsv Product sv
   Product body Iter
       Unavailable;
       Available;
   Getsv Product sv
   Product body End
Product End
```

```
Order Seq
   Read Cust Order;
   Place;
   Read Cust Order;
   Amend Iter (while amendments)
       Amend;
       Write Exception;
        Read Cust Order;
   Amend End
   Deliver or cancel Sel (if cancel)
       Cancel;
        Read Cust Order;
   Deliver or cancel Alt (if deliver)
       Deliver;
        Read Cust Order;
   Deliver or cancel End
Order End
```

```
Order received Seq
   Read Allocation & All Orders;
   Allocate or delay Sel (if sufficient stock)
       Allocate;
   Allocate or delay Alt (if insufficient stock)
       Delay;
   Allocate or delay End
Order received End
```

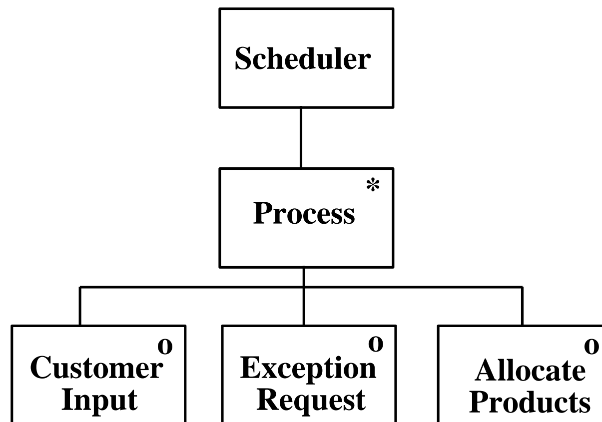
Notes

```
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
      Week 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.

Notes




```
Scheduler Seq
   Read System Input;
   Scheduler body Iter (while input to process)
      Process Sel (if customer input)
         Customer input;
         Loadsv Customer SVFILE;
         Call Customer (customer-id etc.);
         Storesv Customer SVFILE;
      Process Alt (if exception input)
         Exception request;
         Call Exception function;
      Process Alt (if allocation input)
         Allocate products;
         Loadsv Allocation SVFILE;
         Call Allocate function;
         Storesv Allocation SVFILE;
      Process End
   Scheduler body End
Scheduler End
```

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).

7.6. 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.

Notes

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 (structure 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).