The Jackson Development Methods

(Article by Michael Jackson;
in Wiley Encyclopaedia of Software Engineering,
ed J Marciniak, 1992)

JSP and JSD

The Jackson Development Methods are JSP (Jackson Structured Programming) and JSD (Jackson System Development). JSP is a method for designing programs as compositions of sequential processes; JSD is a method for specifying and designing systems whose application domain has a strong temporal flavour and contains objects whose behaviour is describable in terms of sequences of events. Many program and system development problems thus fall within the scope of JSP and JSD, and the methods have been used to develop data processing systems, control systems, systems software, embedded systems, and even a music synthesiser. The UK Government adopted JSP as a standard in 1974, using a slightly modified version known as SDM (Structured Design Method).

JSP and JSD differ from some widely used methods in two main respects. First, they pay attention initially to the domain of the software and only later to the software itself. Their first descriptions describe not the software, but its subject matter. Second, they focus on time-ordering: on event sequencing rather than on static data models. For a JSP program, the domain is the time-ordered streams of records or of events that the program must process. For a JSD system, the domain is the real world in which the entities exhibit concurrent time-ordered behaviours that the system must model and compute about. If JSD entities are regarded as objects, JSD may be said to be object-oriented.

Historically, JSD grew out of JSP; the origins of JSD can be clearly seen in a chapter (Systems and Programs) of the original description of JSP [Jackson 1975]. JSD can be regarded as an extension of JSP to handle larger and more complex compositions of sequential processes, and to address issues of specification and of implementation in typical systems operating environments.

The major sources for JSP are [Jackson 1975], [Ingevaldsson 1979], [Cameron 1983/89], and [Sanden 1985]. The major sources for JSD are [Jackson 1983], [Cameron 1983/89], and [Sutcliffe 1988]. Some works in languages other than English are included among the references at the end of this article. Interesting material on the formalisation of JSP and JSD can be found in [Hughes 1979] and [Sridhar and Hoare 1985]. Suggestions for enhancing the methods in various directions can be found in [Potts et al 1985], [Kato and Morisawa 1987], and [Poo and Lay zell 1990]. A more extensive bibliography is given in [Cameron 1983/89].

THE JSP PROGRAM DESIGN METHOD

Basis of The Method

A program to be designed must process one or more sequential streams of data. The sequentiality of these streams may be spatial or temporal: for example, a stream of data printed on paper or recorded on magnetic tape is spatially sequential, while a stream of messages arriving from an on-line terminal or a stream of interrupts is temporally sequential. The association of a program with its data streams is shown diagrammatically in a System Network Diagram (SND) such as Figure 1. Data streams A, B and C are represented by circles; the program P is represented by a rectangle; the arrows indicate the direction of data flow.

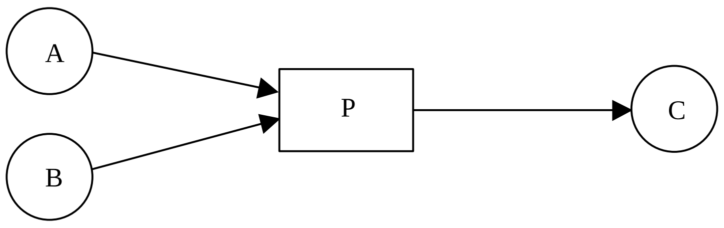

Figure 1. System Network Diagram (SND)

The basic idea is to describe the structures of the sequential streams as regular languages and to form the structure of the program by composing the stream languages into one superset regular language. For example, if the program has an input stream (a(bc)*) and an output stream ((de)*f) and the instances of d and e are constructed from the corresponding instances of b and c respectively, then the program structure is (a((b&d)(c&e))*f) .

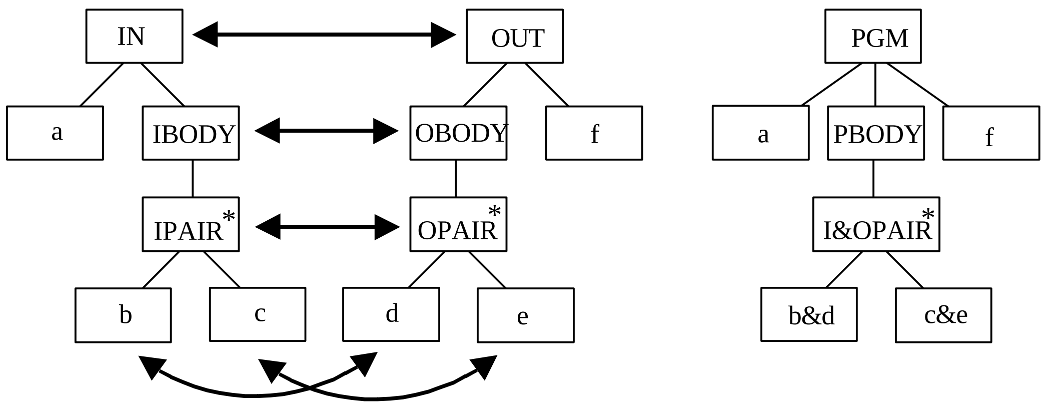

Figure 2. Diagrammatic Notation

This program structure can be thought of as a synchronised traversal of both data structures, in which corresponding items from the two structures (b and d, and c and e) are encountered together. The structures are regular expressions because regular languages provide the three basic constructs — sequence, selection, and iteration — of ‘structured programming’. To describe such regular languages a diagrammatic notation is used. The simple example above is shown in this notation in Figure 2. The two trees on the left of the figure show the data structures; the double-headed arrows between them show the correspondences between their data items; and the tree on the right shows the resulting program structure.

A star in the upper right corner of a box means the same as a star in a regular expression: each IBODY consists of zero or more consecutive IPAIRs.

This technique of program structuring can be compared to the simpler technique of structuring a recursive descent parser according purely to the grammar of the input text to be parsed, and has comparable advantages of clarity and certainty.

The executable operations of the program (comparable to semantic routines in a compiler) can be embedded quite easily in the program structure, because the structure has a program component associated with each component of each data stream. In JSP, determining what executable operations are required and embedding them in the program structure are separate development steps, postponed until the program structure has been designed. Operations must often be associated with a non-leaf component, such as I&OPAIR in Figure 2 above: it is therefore important that the structures should accurately reflect the program functionality as shown by the correspondences.

Reading Ahead and Backtracking

In particular, data and program structures should not be distorted by anticipated difficulty in parsing or navigating a structure. For example, if an input stream has the structure (a((bc)|(bd))*) there is a parsing difficulty when an input b is encountered: is it the first part of a (bc) or is it the first part of a (bd)? The difficulty should not be tackled by rewriting the structure as (a(b(c|d))*) because this rewriting is a distortion, likely to make the structure less clear and the operations harder to embed in the structure. It should be overcome by reading ahead an appropriate number of records in the input stream (the ‘multiple read-ahead’ technique). More difficult cases may demand a backtracking technique, in which program control flow is explicitly modified and the side-effects (of partial traversal of one path that is subsequently rejected in favour of another) are explicitly treated. In this way the problem of dynamic navigation of the structures is separated from the problem of giving their clearest and most appropriate static descriptions.

Structure Clashes: Classification and Resolution

In some cases it is not possible to form a single program structure from the data structures because they do not admit of a single synchronized traversal. In such cases a ‘structure clash’ is said to be present. The program must then be designed as a network of sequential processes rather than as one process.

There are three types of structure clash:

(a) A ‘boundary clash’, in which two structures have groups of data — non-terminal data items — whose boundaries are not synchronised: for example, paragraphs and pages, or months and weeks. Two processes are required, communicating by a stream in which the ‘highest common factor’ items are passed from one to the other.

(b) An ‘ordering clash’, in which corresponding data item instances are differently ordered in two structures. Again, two processes are required, communicating by an intermediate structure in which data can be written in one order and read in the other.

(c) An ‘interleaving clash’, in which groups of data that occur sequentially in one structure are interleaved in another structure. Consider, for example, a chronological list of calls made at a telephone exchange and a report of the same calls arranged chronologically within subscriber. One sequential process is needed for each interleaved group, together with a process to separate out the interleaving and, frequently, another to recombine the output streams from the interleaved group processes.

Structure clashes are, of course, inherent features of the problems in which they occur: they arise from the problem, not from the use of a particular design technique. The contribution of JSP is to show how they may be recognised and resolved.

Program Inversion Technique

Where structure clashes are recognised and resolved, the resulting program consists of two or more sequential processes, each structured to reflect the structures of its data streams. An implementation issue immediately arises: how are the processes to be scheduled? JSP provides an implementation technique, program inversion, for scheduling simple networks of processes communicating by data streams: program inversion can be used with any imperative language that has labels and GOTO statements.

Program inversion allows a sequential process, reading an input stream I and writing an output stream O, to be implemented as any of the following: a main program P; a procedure PI that consumes one record of I on each invocation (and gradually produces the stream O); a procedure PO that produces one record of O on each invocation (and gradually consumes the stream I). The procedure PI is said to be ‘P inverted with respect to I’, and the procedure PO is said to be ‘P inverted with respect to O’. Implementation of inverted programs is discussed in detail in Chapters 8 and 9 of [Jackson 1975] and in Section 3.3 of [Cameron 1983/89]. Essentially, the read or write operations on the relevant data stream are implemented by saving the process state and returning, the next invocation causing a resumption in the saved state.

With program inversion, a pair of communicating processes can be implemented by making one a subroutine of the other. The process that is to become the subroutine is inverted with respect to the data stream by which the processes communicate; the other is implemented as a main program, invoking the subroutine for each record of the stream.

State-Vector Separation

The state of an inverted program — the pointer to the place in the text where execution is to be next resumed and the values of any other local variables of the program — must be preserved from one invocation to the next. In some implementations this can be done implicitly by using ‘static’ or ‘own’ variables; in others a different treatment is required, and the state vector (a data structure containing the program state) must be recognised as a variable in its own right. Like any other variable, it can be passed as a parameter to a procedure, its value can be assigned, and it can be written to backing storage for retrieval at a later time.

State-vector separation is a transformation in which the state-vector of an inverted program is declared as an explicit data structure, and is passed to the procedure as a parameter: one inverted program and many different instances of its state vector can then implement many processes sharing a common definition but having distinct identities and states. This technique allows a natural implementation of the process networks that result from the resolution of interleaving clashes. The processes for the interleaved groups are implemented by inversion and state-vector separation; the main program invokes a particular interleaved process by passing its separated state vector to the inverted program.

FROM JSP TO JSD

Program inversion technique is an important element in the progression from JSP to JSD. Two other important elements are the recognition and resolution of interleaving clashes and an implementation technique — state-vector separation — used to deal with the resulting multiplicity of processes.

Data Processing Systems

A very simple data processing system with one entity type and transactions that update the entity records can be seen as a large program exhibiting an interleaving clash. The input is a chronologically ordered stream of transactions in which the transaction sequence for each entity instance is interleaved with the sequences for the other instances.

Standard resolution of the interleaving clash gives one process to separate out the interleaving and one process for each entity instance. The entity processes are inverted with respect to their input streams of transactions, and their state vectors are separated and stored in the database. The resulting procedure receives a database record (state vector) and transaction (input stream record) as parameters, and updates the database record.

The entity processes are ‘long-running programs’: if the entities are employees, then the elapsed time from the beginning to the end of execution of one entity process will be as long as the employee’s career — probably many years. During most of this long execution time the process is waiting for its next input record: that is, for the next relevant event in the employee’s life. But this is merely a scheduling consideration.

This process-oriented view of the nature of a simple data processing system is developed in Chapter 11 of [Jackson 1975] and summarised in Section 5.1.1 of [Cameron 1983/89].

THE BASIS OF JSD

Modelling the Real World

By setting aside, or at least postponing, consideration of the scheduling of processes on the computer, the process-oriented view derived from JSP highlights the modelling relationship between the system and its application domain or ‘real world’. The processes running in the computer are seen to model the behaviour of entities in the real world. As each relevant event occurs in the life of the real employee, so a record appears in the input stream of the process corresponding to that employee. The JSP data structure of the process input stream is a description of the ordering of events in the employee’s lifetime.

JSD takes this modelling relationship as its starting point. Just as JSP assumes that the program to be designed has sequential streams of input and output data, so JSD assumes that the application domain, or real world, is populated by objects or entities that engage in events that are sequentially ordered in time. Descriptions of the behaviour of these entities will provide the basis for the sequential processes of the system.

System Outputs and Functions

Clearly it is not enough that the system should constitute a model of the real world, kept up to date as real world events occur. It is also necessary to exploit information from the model to provide screen displays, summaries and reports, and to produce outputs such as invoices and payslips for a data processing system or control signals for an embedded system.

JSD regards the production of these outputs as constituting the ‘system function’, and distinguishes them sharply from the ‘real- world model’. The model is regarded as primary, and the functions as secondary: the model is defined first, and the functions are then defined on the basis of the model. This approach is common in simulation, where the simulation of reality is defined first, and then a variety of reporting and analysis functions are added later.

Relationship of Model to Function

The underlying relationship of a system’s functions to its model of the real world is, of course, circular. In principle it is possible, given a model, to determine what functions it can support; in principle it is also possible, given a set of functions, to determine what model is needed to support them. But there are good reasons to regard the model as primary and the functions as secondary. First, defining functions intelligibly requires the use of terms taken from the real world. Second, the set of functions required of a system is much more volatile than the underlying real-world model. Third, the real-world model can be regarded as itself defining an envelope of all possible system functions: the system can provide those functions for which the necessary information is available in the model, and no others.

Data Models and Event Models

The idea of modelling the real world is not unique to JSD. But JSD is unusual in two respects: it insists on a rigorous separation of model from function, expelling from the model any events or entities that properly belong to the system function; and its models are based on events and sequential processes rather than on data. Data models concentrate on the states of the modelled domain; event and process models concentrate on the way in which those states change.

In JSD a data model is largely derived from the event and process model. This is appropriate for the majority of applications in which describing the dynamic time-ordered behaviour of entities captures a larger part of the relevant properties of the real world than describing the system state.

JSD DEVELOPMENT STEPS: INTRODUCTION

The development steps of JSD as originally formulated in [Jackson 1983] are six: the entity action step; the entity structure step; the initial model step; the function step; the system timing step; and the implementation step. As described in [Cameron 1986] and [Cameron 1988] these steps were later modified and rearranged into three development phases:

- The modelling phase, in which the real world is described in terms of events, entities, roles, event orderings, and entity attributes.
- The network phase, in which the system processes are configured into a process network, beginning with the model processes and adding other processes to collect input data, produce outputs, and generate certain model inputs.

- The implementation phase, in which timing is considered, the implementation and scheduling of the process network are decided, and the database, if there is one, is designed.

The exposition in this section follows the broad lines of this later formulation, and also draws on some later refinements and elaborations of the method.

THE MODELLING PHASE

Identifying Real-World Events

JSD modelling begins by identifying the real world events (also called actions) that are of interest. For each event type a name is chosen, an informal description is written by which occurrences of the event type in the real world may be recognised, and the attributes of the event are listed. For example, in a library application, the event list may include:

ACQUIRE	The library acquires a book.	(book-id, date, ISBN)
CLASSIFY	A book is classified.	(book-id, class)
JOIN	A new member joins the library.	(member-id, name, address, date)
LEAVE	A member leaves the library.	(member-id, date)
BORROW	A member borrows a book from the library.	(book-id, date, member-id)
RETURN	The borrowing member returns a book to the library.	(book-id, date, member-id)
SELL	The library sells one of its books.	(book-id, date, price)

Each event is something that happens in the real world, not in the computer system itself; it is regarded as atomic; it is not decomposable into sub-events; and it is considered to happen instantaneously.

In a realistic application there may be a hundred or more event types in the list. The event list is a first step in defining the scope of the system. It includes only those real-world events of which the system will take cognisance, and so acts as a filter through which the system views and formalises the real world.

Identifying Real-World Entities and Structures

In the original version of JSD, identification of real-world entities was roughly on a par with identification of events: there was an event list and an entity list. Because the entities of most interest in JSD development were those that have a significant time-ordered history of events, it was soon recognised that their identification could be combined with defining the structures of event orderings.

For example, in the library application, books and members are obvious entities of interest. But if this were not immediately clear, it would emerge from a consideration of the ordering of events: the ordered subsets of all events are those in which one particular book participates and those in which one particular member participates. These orderings are shown in two JSP structures in Figure 3.

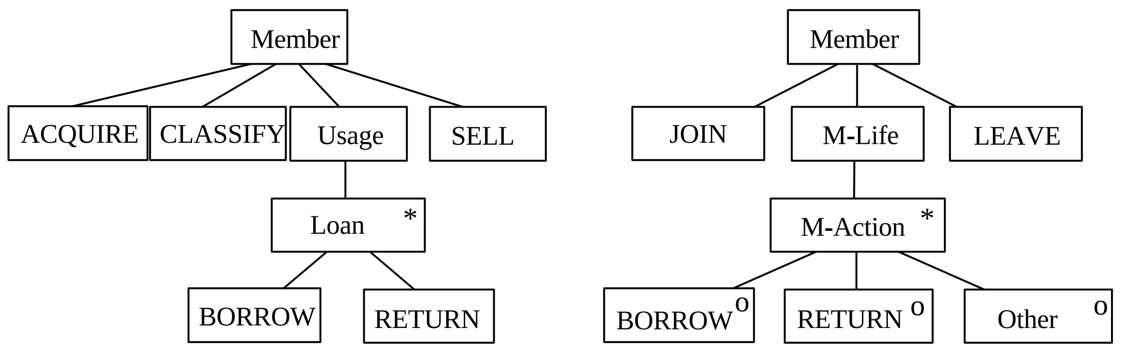

Figure 3. Ordering of Events

The circles in the upper right corners of BORROW, RETURN and Other in the Member structure in Figure 3 indicate that they are parts of a selection: each M-action is either a BORROW, or a RETURN, or an Other. These two structures describe the concurrent behaviour of the many books and the many members of the library.

Entity Attributes

As an entity progresses through its event history its state changes. At one time the book is in the library, at another it is not; at one time it has not yet been classified, then it has; at one time it is on loan to a particular member, at another time to another member, and at yet another time it is not on loan to any member.

Such a set of states defines the attributes of an entity. A book has a ‘current borrower’ attribute; initially, at the beginning of the book’s life, its value is NULL; when the book is borrowed by member M the attribute value becomes M; when the book is returned it becomes NULL again. The book attribute ‘number of loans’ is initially zero; it may be incremented on each BORROW event, or, alternatively, on each RETURN event.

This second example illustrates an important advantage of defining entity attributes in this way, on the basis of a time-ordered structure of events: the meanings of the attribute values can be made fully unambiguous. The attribute ‘number of loans’ could reasonably mean either ‘number of completed loans’ or ‘number of started loans’: by explaining the values in terms of the events by which they are changed, misunderstanding is avoided.

This derivation of entity attributes from events may be seen as a second step in defining the scope of the system. The event list determines what events are relevant; the entity attributes determine what is remembered about those events.

Common Actions

Some events, such as CLASSIFY, involve only one entity. Others, such as BORROW, involve both a member entity and a book entity. In this way the set of entity structures can define more complex constraints on the ordering of events than can be defined by one structure alone: the BORROWing of a book occurs only if the member has previously JOINed and has not yet performed a LEAVE event, and only if the book’s most recent event was either CLASSIFY or RETURN.

Data Models

The common event BORROW and the ‘current borrower’ attribute illustrate the basis of the association between a data model and an event model. In an entity-relation model, for example, there would be member and book entities, and a relation ‘is borrowing’ between them. This relation is reflected in the event model by the fact that the value of the ‘current borrower’ attribute of the book entity is an identifier of a member entity; the value is derived from the occurrence of a common event in which the book and the member both participated. Consideration of the common events and the attributes derived from them allows a data model to be obtained from a JSD event model.

Multiple Roles

The event ordering for one entity type may not be completely describable in one structure. In general, an entity may play more than one role in the real world. For example, a vehicle owned by an organisation may have two roles: in the first, operational, role the events concern assignments to particular drivers, periods of use on projects, incidents such as accidents, and repairs and regular servicing; in the second, financial, role the events concern purchase, sale, licensing, disbursement of running costs, and depreciation (amortisation). These different roles will require to be kept separate both on general grounds of application modularity and because they are likely to exhibit boundary clashes: for example, there is probably a boundary clash between the periods of project use and the licensing periods. Often, distinct roles of an entity will have common actions, reflecting the synchronisation between different aspects of the entity’s behaviour. The association of several roles with one entity type and the sharing of roles among entity types may be compared to the treatment of object properties by class-based inheritance.

Static Roles

Viewing entity attributes as components of process states, their values definable in terms of event occurrences, is an important insight. But not all relevant attributes can usefully be regarded in this way. An employee’s date of birth or marital status is clearly definable in terms of events, but those events are hardly within the scope of a payroll system. To handle such attributes it is useful to allow ‘static’ roles, whose attributes are either fixed by events outside the scope of the system or are changed by arbitrary replacement of their values: the replacement events are considered to be unconstrained in the order of their occurrence with respect to each other and to other events.

THE NETWORK PHASE: PROCESSES AND CONNECTIONS

In the modelling phase the work centres on describing the real world in terms of entities, events, and entity and event at tributes. So far no explicit attention has been paid to the computer system itself. In the network phase a set of sequential processes is defined that models the real world; this set is then elaborated into a process network by the addition of processes for handling input messages from the real world and for generating system outputs.

The System Specification Diagram

The connections among processes in the network are shown in a System Specification Diagram or SSD: this is a much more elaborate version of the System Network Diagram of JSP. The processes in a SSD are considered to be executing asynchronously. The types of connection are illustrated in Figure 4. Each set of processes of a given type is represented by a rectangle (P, Q, R and S). There are three forms of process communication: data stream connection, represented by a circle (A, B, C, D, E and G); controlled data stream connection, represented by a circle with a vertical bar (F); and state-vector inspection, represented by a diamond (Q). Decomposition of SSDs by levels is little used; for particular purposes, partial SSDs are drawn containing only those processes and connections that are of current interest.

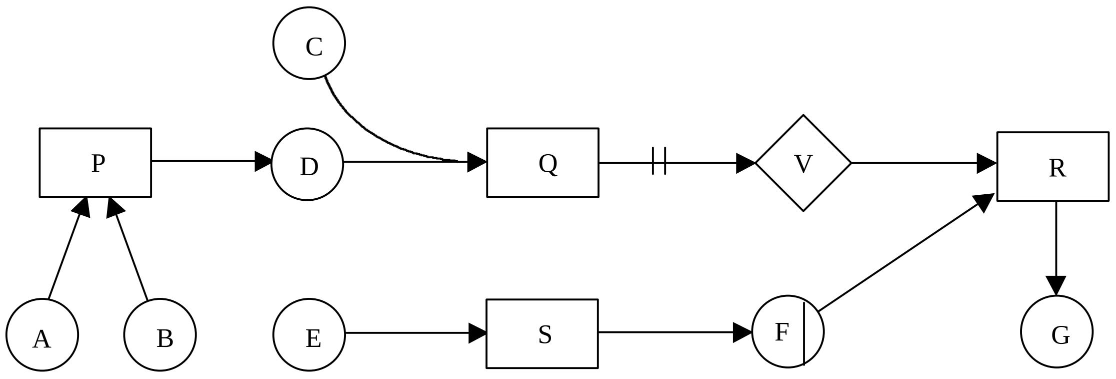

Figure 4. System Specification Diagram (SSD)

Data Stream Communication

A data stream is an unbounded queue. The read operation on a data stream retrieves (and removes) the next record from the queue. If the queue is empty then the process executing the read operation waits until a record becomes available. The write operation on a data stream places the next record in the queue. Since the queue is unbounded the process executing the write operation is never forced to wait.

State-Vector Inspection

A state-vector inspection is a ‘getsv’ operation in which the executing process is granted read-only access to the state vector of the inspected process. The getsv operation never causes the inspecting process to wait. To the inspected process a getsv operation is invisible: the inspected process executes no operation corresponding to the getsv, and no test is available to it by which it could detect that its state has been inspected.

The value of the state-vector obtained by a getsv operation represents a current or previous state of the inspected process, at a point at which its execution is suspended. Just as the reading process in a data stream connection may lag behind the writing process, so the process executing a getsv operation may lag behind the inspected process, retrieving values that represent states much earlier than its current state.

Rough Merges

A process with more than one input stream may be waiting to read a record from one stream while a record is already available on another. No provision is made for a process to test whether an input stream is currently empty. Instead, two or more input streams to a process may be merged into one by an implicit process called a ‘rough merge’. In a SSD a rough merge is represented by merging the outgoing arrows of the streams: thus in Figure 4 the input streams C and D of process Q are rough merged, while the input streams A and B of process P are not. From the point of view of the reading process Q, the streams C and D constitute a single merged stream C&D.

Rough merge is non-deterministic: the order of arrival of C and D records at the reading process Q will, in general, depend on the scheduling and execution speeds of the writing processes, and these are not defined until the implementation phase. The structure of a merged data stream must, therefore, describe all possible mergings.

Multiplicity

It will often be the case that a process communicates with many instances of another process. The arrows on data stream and state-vector connections in the SSD are decorated with double bars to show such multiplicity. In Figure 4, the process R inspects the state vectors of many instances of the process Q.

Controlled Data Streams

The scheduling freedom that lies behind the process communication operations is usually beneficial: the many processes whose concurrent execution constitutes the system behaviour are primarily asynchronous. Sometimes, however, a tighter form of communication is necessary: for these cases a special form of communication is provided: controlled data stream communication. In Figure 4 the process R reads a controlled data stream F written by process S. Controlled data streams combine data stream communication with the ability for the data stream writer to examine the current state of the reader and with a form of critical region.

THE NETWORK PHASE: MODEL AND FUNCTION

Model and Input Processes

The description of the real world that has been made in the Modelling Phase allows a very direct simulation by sequential processes within the system. Essentially each role or entity becomes an executable process in the system, consuming an input stream that contains a record for each relevant event in the real world. All data are local to these processes, which may be regarded as encapsulated objects.

In addition to these entity or role processes, the system must contain processes to distribute the messages coming from the real world to the system processes to which they are relevant. This distribution must ensure not only that messages are routed to the appropriate processes but also that messages are not sent to processes that are not able to handle them: a message for a common action must be sent to all the relevant processes or to none. The input processes must therefore check the acceptability of each message before routing it to a model process: the con trolled data stream provides the necessary form of communication.

Embedded Functions

Some of the output functions of the system can be added directly to model processes. For example, if a simple acknowledgement of a BORROW event is required, it is straightforward to embed the production of the necessary information in the book model process. Such simple functions are called ‘embedded functions’. They are usually associated with the occurrence of a single event and require information that is to be found in the state of a single model process.

Output Functions with State-Vector Connection

Many output functions can be provided by the addition of one process that is connected to the model by state-vector inspection: the function process has an input stream of query messages, and for each query it obtains its output information by examining the state of the model. In a banking system a function to query the balance on an account would probably be of this kind: in response to an input message containing an account number, it inspects the state vector of the account process and outputs the value of the balance variable in the state vector at some indeterminately recent time. How recent will depend on the implementation of the system: in a fully on-line system the value will be at worst a second or two out of date; in a system with daily batch processing components the value may be out of date by a whole day. The implementation must be decided eventually, but not yet.

Output Functions with Data Stream Connections

Some output functions must summarise sequences of events from many model processes. For example, the library may require a daily list of all BORROWings. In a library that is always open, BORROWings that occur at a time close to the time of producing a daily list may appear on that list or the next: this indeterminacy arises from the indeterminate scheduling of the model and function processes. However, it must not be possible for any BORROWing to be altogether omitted, and to ensure this it is necessary to use a data stream connection as in Figure 5.

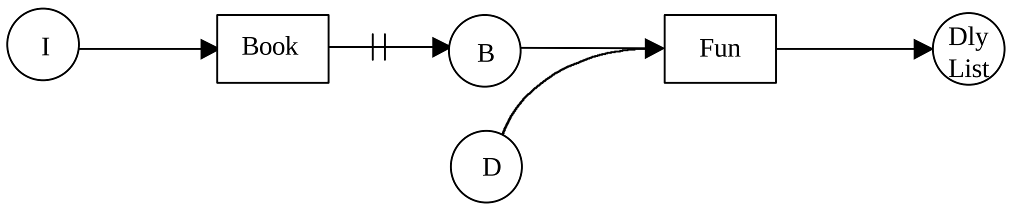

Figure 5. Data Stream Connections

Each book process writes a record on an output stream B for each BORROW event; these streams B are rough merged with each other and with the messages D that indicate that another day’s list is required. The function process F reads this merged stream and produces one day’s list whenever it encounters a D message. Extreme choices of process scheduling may give undesirable partitioning of the total history of BORROW events into the daily lists, but can not cause any BORROW event to be omitted.

Interactive Functions

For the model processes to simulate the real world correctly, it is necessary that they should receive input messages informing them of all occurrences of relevant real-world events. Sometimes it may be difficult, expensive, or even impossible to arrange such inputs directly from the real-world, yet the event occurrences may be deducible from events already known to the system. For example, if a library member LEAVEs, it may be inferred that all outstanding book reservations for that member are cancelled.

Messages for such inferred events are provided by ‘interactive function’ processes. These are simply processes whose outputs include data streams that are fed back into model processes, where they are rough merged with the data streams coming from the input processes.

IMPLEMENTATION PHASE

Implementation of a JSD specification is concerned with two main issues: how the processes of the specification are to be scheduled, and how the data — the process state vectors — are to be organised and managed. The techniques used are essentially extensions of those already familiar from JSP.

Channel Inversion

Program inversion is extended to allow inversion with respect to more than one data stream. For example, a process P with an input stream I and an output stream O can be inverted with respect to both I and O, giving a procedure that may be invoked either to consume a record of I or to produce a record of O. In certain special cases the pattern of invocations is predetermined. For example, P may read from I and write to O alternately: this pattern reflects a ‘conversational constraint’ on the communication, and is easily implementable by the usual mechanism of procedure call with input and output parameters. But in general the invoking program must be able to determine which of these operations P is ready to execute.

Explicit Schedulers

In JSP implementation of a program containing more than one process the standard approach is to transform the set of processes into one sequential process, using program inversion and state-vector separation. Almost always, one of the processes of the network can be chosen as the main program, invoking all other processes directly or indirectly; the scheduling of the network is then expressed implicitly in the inversion choices made.

In many systems, especially in control applications, a similar approach may still be used. In others, especially data processing systems, the process network is much more complex and the timing requirements much more detailed: some processes must be kept as nearly as possible up to date with the real world objects they model, while others may be allowed to lag behind far enough to permit a batch implementation of a part of the system. To handle these more complex demands it is usually necessary to provide explicit scheduling processes. These schedulers are themselves long-running programs, and will require transformation for execution on a machine that is not always running and must be shared with other application systems.

Buffering

Networks containing cycles will, in general, require buffering of data streams between processes. The transaction files that abound in systems with batch processing components can be viewed as buffered data streams. Typical batch update programs can be viewed as parts of scheduling processes invoking inverted model processes to update the master records.

Data Base Implementation

The usual techniques of data base design, especially for relational data bases, are likely to demand a reorganisation of the process state vectors. A state vector containing repeated groups may be normalised and split into several data base records or tuples; state vectors of different processes, especially different roles of one entity, may be combined into one data base record.

Access paths to data base records are explicit in the process texts of function processes that use state-vector inspection. Optimisation of the data base organisation for these access paths is a matter of finding a single linked structure of which each such access path is a subset. The problem is analogous to, but different from, the problem of finding a program structure that matches each of several data structures in JSP design.

REFERENCES

[Andr'as 1983] M Andr'as; Programtervez'es Jackson-m'odzerrel; Computing Applications and Service Company, Budapest; 1983 (Hungarian).

[Burgess 1984] R S Burgess; An Introduction to Program Design Using JSP; Hutchinson, London; 1984.

[Cameron 1983/89] J R Cameron; JSP & JSD: The Jackson Approach to Software Development (1st edition 1983, 2nd edition 1989); IEEE CS Press, Washington DC; 1983, 1989.

[Cameron 1986] J R Cameron; An Overview of JSD; IEEE Transactions on Software Engineering, Volume 12 Number 2; pp 222-240; February 1986. (Reprinted in Cameron 1983/89, 2nd edition)

[Cameron 1988] J R Cameron; The Modelling Phase of JSD; Information and Software Technology, Volume 30 Number 6; pp 373-383; July/August 1988.

[Hughes 1979] J W Hughes; A Formalisation and Explication of the Michael Jackson Method of Program Design; Software Practice and Experience, Volume 9 pp 191-202; 1979.

[Ingevaldsson 1979] Leif Ingevaldsson; JSP: A Practical Method of Program Design; Input-Two-Nine, London; 1979.

[Ingevaldsson 1985] Leif Ingevaldsson; JSD — metoden for systemutveckling; Studentlitteratur, Lund; 1985.

[Jackson 1975] M A Jackson; Principles of Program Design; Academic Press, London; 1975. (Also translations in Dutch, German, Japanese, Portuguese, and Spanish.)

[Jackson 1976] M A Jackson; Constructive Methods of Program Design; in Proceedings of the 1st ECI Conference 1976, pp 236-262; Springer Verlag LNCS 44, Heidelberg 1976. (Reprinted in Cameron 1983/89)

[Jackson 1978] M A Jackson; Information Systems: Modelling, Sequencing, and Transformations; in Proceedings of the 3rd International Conference on Software Engineering, pp 72-81; IEEE CS Press, Washington DC; 1978.

[Jackson 1983] M A Jackson; System Development; Prentice-Hall International, London; 1983. (Also translations in Dutch and Japanese.)

[Jansen 1983] Henk Jansen; Jackson struktureel programmeren; Academic Service, Arnhem; 1983 (Dutch).

[Josephs 1989] Mark B Josephs, C A R Hoare and He Jifeng; A Theory of Asynchronous Processes; Oxford University Computing Laboratory Technical Report PRG-TR-6-1989.

[Kato and Morisawa 1987] J Kato and Y Morisawa; Direct Execution of a JSD Specification; in Proceedings of COMPSAC 11; IEEE CS Press, Washington DC; 1987.

[Kilberth 1988] Klaus Kilberth; Einführung in die Methode des Jackson Structured Programming; Vieweg & Sohn, Braunschweig; 1988.

[King 1988] D King; Creating Effective Software: Computer Program Design Using the Jackson Methodology; Yourdon Press, USA; 1988.

[Poo and Layzell 1990] C-C D Poo and P J Layzell; Enhancing the Software Maintenance Factor in JSD Using Rules; in Proceedings of CompEuro'90, pp 218-224; IEEE CS Press, Washington DC; 1990.

[Potts et al 1985] C Potts, A Bartlett, B Cherrie, and R McLean; Discrete Event Simulation as a Means of Validating JSD Specifications; in Proceedings of the 8th ICSE; IEEE CS Press, Washington DC; 1985.

[Sanden 1985] Bo Sanden; Systems Programming with JSP; Chartwell-Bratt, Bromley; 1985.

[Sridhar and Hoare 1985] K T Sridhar and C A R Hoare; JSD Expressed in CSP; Oxford University Computing Laboratory Technical Monograph PRG-51; 1985.

[Sutcliff 1988] A Sutcliffe; Jackson System Development; Prentice-Hall International, London; 1988.

[Thompson 1989] J B Thompson; Structured Programming with COBOL and JSP (2 Volumes); Chartwell-Bratt, Bromley; 1989.