An Overview of JSD

JOHN R. CAMERON

***Abstract*—The Jackson System Development (JSD) method addresses most of the software lifecycle. JSD specifications consist mainly of a distributed network of processes that communicate by message-passing and by read-only inspection of each other's data. A JSD specification is therefore directly executable, at least in principle. Specifications are developed middle-out from an initial set of "model" processes. The model processes define a set of events, which limit the scope of the system, define its semantics, and form the basis for defining data and outputs. Implementation often involves reconfiguring or transforming the network to run on a smaller number of real or virtual processors. The main phases of JSD are introduced and illustrated by a small example system. The rationale for the approach is also discussed.**

***Index Terms*—Design methodology, system design, systems analysis.**

I. Introduction

THE Jackson System Development (JSD) approach aims to address most of the software lifecycle either directly or by providing a framework into which more specialized techniques can fit. JSD can be used from the stage in a project when there is only a general statement of requirements right through to the finished system and its subsequent maintenance. Many projects that have used JSD actually started slightly later in the lifecycle, doing the first steps largely from existing documents rather than directly with the users.

A JSD specification consists (mainly) of a distributed network of sequential processes. Each process can contain its own local data. The processes communicate by reading and writing messages and by read-only access to one another's data. The specification is developed middle-out starting with a particular set of "model" processes. Most of the data in the system belongs to these model processes. New processes are added to the specification by connecting them to the model. Usually the only direct connections in a network are between model and nonmodel processes and between nonmodel processes and the outside. Thus one model process is not directly connected to another or to the network boundary, and nonmodel processes only interact via the model processes. The exceptions to this general rule are discussed in Section V-B.

Direct implementations of this executable network are possible in principle and sometimes also in practice. Often, however, the network is reconfigured during the implementation phase by mapping the specification processes on to a smaller number (perhaps one) of implementation processes. The reconfiguration involves fixing some of the scheduling that was left relatively unconstrained in the specification network. The other major concern in the implementation phase is the choice of storage structures (physical database design in an information system) to hold the data owned by the processes. The storage structures must also support the read-only access requirements of the other processes.

There are three main phases in the JSD method, the Model phase in which the model processes are selected and defined, the Network phase in which the rest of the specification is developed, and the Implementation phase in which the processes and their data are fitted on to the available processors and memory.

Sections II, III, and IV of this paper are concerned with, respectively, the model, network, and implementation phases. Section V contains the following five topics:

- A comparison of the JSD modeling approach to a more functional view of systems.
- A discussion of composition and of decomposition as general development strategies.
- A discussion of the variety of ways that the JSD steps can be mapped into the managerial framework of a project plan.
- A brief description of projects that are using or have used JSD.
- A brief description of available support tools.

II. The Modeling Phase

A. A Model with Only One Process Type

The modeling phase is concerned first with "actions" (or "events") about which the system has to produce displays, reports, signals, and other outputs. For most systems these events are mainly to be found in the world external to the system being built. They are selected and defined along with their associated attributes, and their mutual orderings described by a number of sequential processes.

Fig. 1, for example, is a list of eight actions taken from a simplified library system. By choosing these actions, and only these, we are defining a first scoping of the system. By "scoping" we mean a (indirect) definition of the range of functionality of the system. Later in the development a detailed choice will be made from this range.

We may imagine a pair of spectacles through which only the selected actions can be observed. The system to be built is like a person wearing these spectacles; its outputs can only be based on what has been observed of the world.

Manuscript received April 8, 1985.
The author is with Michael Jackson Systems, Limited, London W1N 5AF England.
IEEE Log Number 8405410.

Action	Definition and Attributes
ACQUIRE	The library acquires the book. id, date, title, author, ISBN, price
CLASSIFY	The book is classified and catalogued. id, date
LEND	Someone borrows the book. id, date, borrower
RENEW	The borrower renews the loan. id, date
RETURN	The borrower returns the book to the library. id, date.
SELL	The book is sold. id, date, vendor, price
OUTCIRC	The book is taken out of circulation as part of the inter-library swap scheme. id, date, destination
DELIVER	The book is delivered to the other library. id, date

Fig. 1.

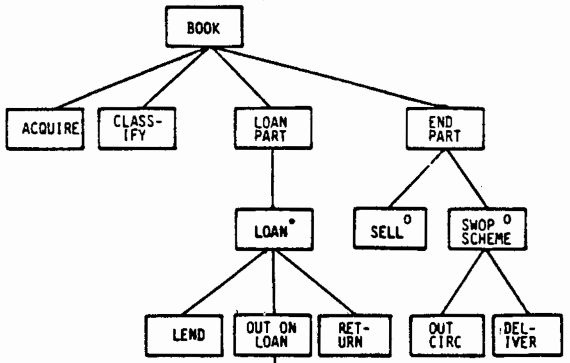


Fig. 2.

The diagram in Fig. 2 describes for one book the order in which the actions can happen. The diagram is a tree structure; the leaves are the actions; all the other components describe sequential relationships between actions or between groups of actions. Excepting the leaves there are three types of component—sequence, iteration, and selection. BOOK is a sequence of ACQUIRE, CLASSIFY, LOANPART, and ENDPART. That means that BOOK consists of one ACQUIRE, followed by one CLASSIFY, followed by one LOANSET, followed by one ENDPART. Similarly LOAN is a sequence of one LEND, then one OUTONLOAN, and then one RETURN.

LOANPART is an iteration of LOAN. That means LOANPART consists of zero or more LOAN's, one after the other. (An iteration is a generalization of a sequence.) Similarly, OUTONLOAN consists of zero or more RENEW's.

ENDPART is a selection component. That means ENDPART consists of either exactly one SELL or exactly one SWOPSCHEME. Sequences and selections can be of two or more parts; iterations can only be of one. Iterations and selections are denoted, respectively, by the "*" and "o" in the top right corner of their constituent components. (Logically, but somehow not to the eye, the identifying symbol is in the wrong box.)

We are, of course, using a diagrammatic notation for regular expressions. We also use recursion within the diagrams, where appropriate.

Fig. 2 describes a set of sequences of actions. The following are two members of the set, two possible life histories for a book.

ACQUIRE, CLASSIFY, LEND, RETURN, LEND, RENEW, RETURN, SELL.
ACQUIRE, CLASSIFY, LEND, RENEW, RENEW, RETURN, OUTCIRC, DELIVER.

The complement of the set of sequences describes, by implication, what cannot happen. A LEND cannot immediately follow an ACQUIRE; a SELL cannot immediately follow a LEND. Later this information will be used to build some of the error-handling parts of the system.

If an input suggests that a book has been SOLD immediately after a LEND, we know that there has been some error on input because in accepting this diagram we are agreeing that a SELL cannot follow a LEND without an intervening RETURN.

The diagram describes orderings but it says nothing about how much time elapses between successive actions.

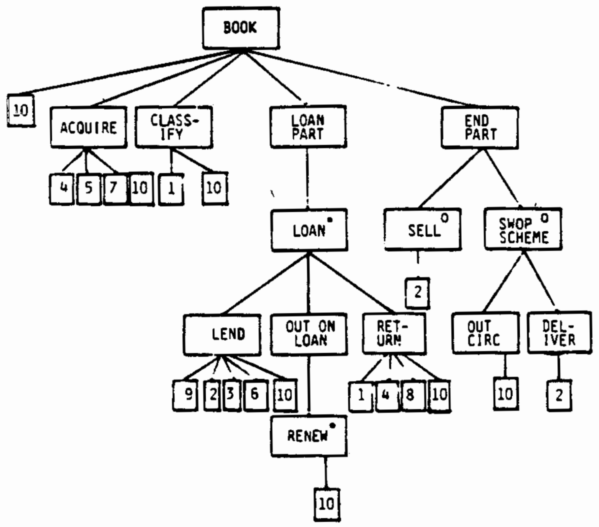
Each diagram describes the actions of one book. To describe the many books in the library we must have many instances of the diagram.

So far the diagram describes the library itself. Now we are also going to use the same diagram to describe a process type within our specification. The name of the process type is BOOK; there will be one instance for each book in the library; the purpose of a BOOK process is to model or mimic what is happening to the real book outside; to this end the process reads inputs, one for each action; the purpose of the input is to inform the BOOK model of what has happened so that the model can coordinate itself with the reality. A textual form of the process is shown in Fig. 3. This pseudocode is simply a transcription of the diagram with conditions added and reads inserted according to a read-ahead scheme. Note that one instance of this process will take as long to execute as the corresponding book is in the library. If we were able to observe the state of the partially excuted process we would know something, but not much, about the state of the corresponding book.

Having described what happens in the library, and built a process model that keeps track of what happens, we are in a position to define data. The diagrams and equivalent text in Figs. 4 and 5 define some example data items: INLIB, ONLOAN, LOANCT, TIMEONLOAN, and LOANDATE. Ignoring some of the technical details, the important points are as follows: the original model process is used as a framework for defining the data to be stored for one book; the meaning of the data is formally tied to the meaning of the actions and their attributes; a data item is local to a process instance; the mechanism for updating

```
BOOK seq
  read next input ;
  ACQUIRE seq
    read next input ;
  ACQUIRE end
  CLASSIFY seq
    read next input ;
  CLASSIFY end
  LOANPART iter  while (input = LEND)
    LOAN seq
      LEND seq
        read next input ;
      LEND end
      OUT-ON-LOAN iter while (input = RENEW)
        RENEW seq
          read next input ;
        RENEW end
      OUT-ON-LOAN end
      RETURN seq
        read next input ;
      RETURN end
    LOAN end
  LOANPART end
  ENDPART select (input = SELL)
    SELL seq
    SELL end
  ENDPART alt (input = OUTCIRC)
    SHOPSCHEME seq
      read next input ;
    SHOPSCHEME end
    DELIVER seq
    DELIVER end
  ENDPART end
BOOK end
```

Fig. 3.



```
1.  INLIB := 'Y'                7.  TIMEONLOAN := 0
2.  INLIB := 'N'                8.  TIMEONLOAN := TIMEONLOAN
3.  ONLOAN := 'Y'                     + IN-DATE  - LOAN-DATE
4.  ONLOAN := 'N'               9.  LOAN-DATE := IN-DATE
5.  LOANCT := 0                 10. READ NEXT INPUT
6.  LOANCT := LOANCT + 1
```

Fig. 4.

the data is part of its definition, not something separate; the definition is in terms of event histories; as the model process executes to keep in step with the reality the data is also kept up to date; for this reason we have avoided problems of data integrity.

Defining these data begins a second, more restrictive scoping of the specification. The actions define what happens, the data define what is to be remembered about what has happened. The system can only use historical data in its outputs or in the conditions for producing outputs if that data has been stored. This also applies to simple items

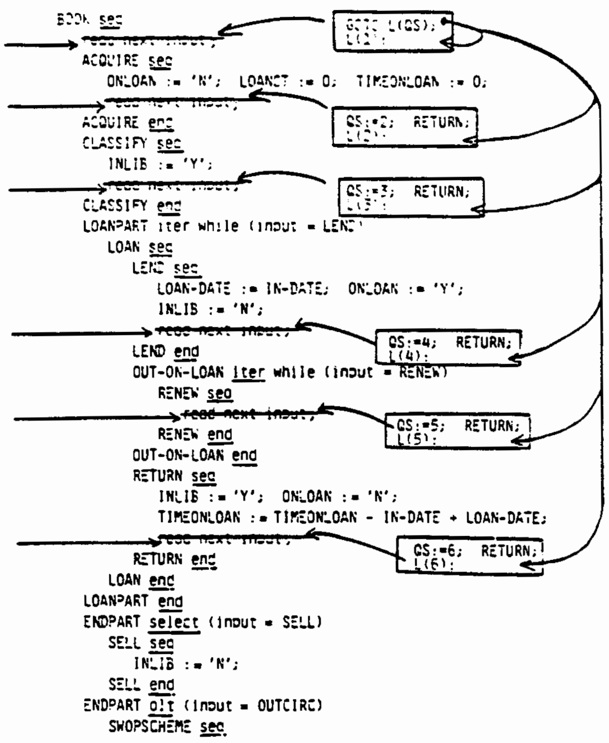
```
BOOK seq
  read next input;
  ACQUIRE seq
    ONLOAN := 'N';  LOANCT := 0;  TIMEONLOAN := 0;
    read next input;
  ACQUIRE end
  CLASSIFY seq
    INLIB := 'Y';
    read next input;
  CLASSIFY end
  LOANPART iter while (input = LEND)
    LOAN seq
      LEND seq
        LOAN-DATE := IN-DATE;  ONLOAN := 'Y';
        INLIB := 'N';
        read next input;
      LEND end
      OUT-ON-LOAN iter while (input = RENEW)
        RENEW seq
          read next input;
        RENEW end
      OUT-ON-LOAN end
      RETURN seq
        INLIB := 'Y';  ONLOAN := 'N';
        TIMEONLOAN := TIMEONLOAN - IN-DATE + LOAN-DATE;
        read next input;
      RETURN end
    LOAN end
  LOANPART end
  ENDPART select (input = SELL)
    SELL seq
      INLIB := 'N';
    SELL end
  ENDPART alt (input = OUTCIRC)
    SHOPSCHEME seq
      •
      •
      •
      •
      •
```

Fig. 5.

like ACQUIRE-DATE, ISBN, and TITLE which are attributes of the ACQUIRE action and therefore are part of the ACQUIRE input transaction. Simple operations are needed in the BOOK process to store these data items, if they are required. (So far we have only shown how to define data in model processes; other processes may contain data: for example, we may introduce a process to hold the total numbers and values of books acquired in each of the last five weeks, or a process for each author to accumulate the number of LEND's in successive periods. These extra data are also part of the second scoping. Nevertheless, unless we remember every attribute of every action, the second scoping will be more restrictive than the first.)

For obvious reasons, we sometimes describe processes like the BOOK process as long-running processes. Only in some environments can such processes be executed directly. In others, an explicit suspend-and-resume mechanism has to be introduced. It could work as follows. When the process reaches a read it suspends itself; when a record becomes available, possibly several months later, the process resumes where it left off, executes as far as the next read, and suspends itself again. Between suspension and resumption, the values of any local variables and the resume point in the program (collectively called its "state vector") must be stored explicitly on some file or database. From the JSD point of view, the files or database of a system are simply the state vectors of the partially executed long-running processes that make up the specification.

A possible suspend-and-resume mechanism is illus-



```
BOOK seq
    ~~read next input;~~
    ACQUIRE seq
        ONLOAN := 'N';  LOANCT := 0;  TIMEONLOAN := 0;
    ~~read next input;~~
    ACQUIRE end
    CLASSIFY seq
        INLIB := 'Y';
    ~~read next input;~~
    CLASSIFY end
    LOANPART iter while (input = LEND)
        LOAN seq
            LEND seq
                LOAN-DATE := IN-DATE;  ONLOAN := 'Y';
                INLIB := 'N';
                ~~read next input;~~
            LEND end
            OUT-ON-LOAN iter while (input = RENEW)
                RENEW seq
                    ~~read next input;~~
                RENEW end
            OUT-ON-LOAN end
            RETURN seq
                INLIB := 'Y';  ONLOAN := 'N';
                TIMEONLOAN := TIMEONLOAN - IN-DATE + LOAN-DATE;
                ~~read next input;~~
            RETURN end
        LOAN end
    LOANPART end
    ENDPART select (input = SELL)
        SELL seq
            INLIB := 'N';
        SELL end
    ENDPART alt (input = OUTCIRC)
        SWOPSCHEME seq
```

```
GOTO L(QS);
L(1):

QS:=2;  RETURN;
L(2):

QS:=3;  RETURN;
L(3):

QS:=4;  RETURN;
L(4):

QS:=5;  RETURN;
L(5):

QS:=6;  RETURN;
L(6):
```

Fig. 6.

trated in Fig. 6. Some details are omitted. For example, QS must be initialized before the first call, either by declaration or perhaps by the calling program. This mechanism converts the BOOK process into a BOOK subroutine. If several processes are similarly converted into subroutines of the same main program, then the several specification process will have been combined into a single implementation process.

The diagrams used in the first instance simply to describe and analyze the library later become part of the code of the final system. Model processes can be turned into update subroutines for the database, files, or other stored data.

We have digressed into a discussion of some issues to do with the implementation of model processes. We now return to the modeling phase and consider some more complicated models.

B. *More Complicated Models*

Fig. 7 introduces five new actions to the library system and discriminates between two cases of an existing action. There can be several copies of the same book, so a RESERVE refers to a title not to an individual book. Fig. 8 shows two more structures that describe the orderings of the actions, making three structures in all.

Several of the actions belong to more than one of the structures. The different structures describe intersecting subsets of actions; each structure describes a set of ordering constraints on its own actions. Thus we can view the same events in the same reality from different points of view. The constraints on any one action are the sum of the

A MORE COMPLICATED MODEL FOR THE LIBRARY

More Actions

Action	Definition and Attributes
JOIN	A new member joins the library member-id, name, address, lend-limit, date.
LEAVE	A member leaves the library, or through inactivity is deemed to have left. member-id, date.
CHANGE-DETAILS	A member's address, lend-limit or reserve-limit is changed. member-id, address, lend-limit, reserve-limit
RESERVE	A member reserves a title that isn't available member-id, title
CANCEL	A reservation is no longer wanted. member-id, title

In addition we have sometimes to distinguish two kinds of LEND action:

LEND-RESERVE	The LEND is the result of a reservation
LEND-NORMAL	The LEND is not the result of a reservation

Fig. 7.

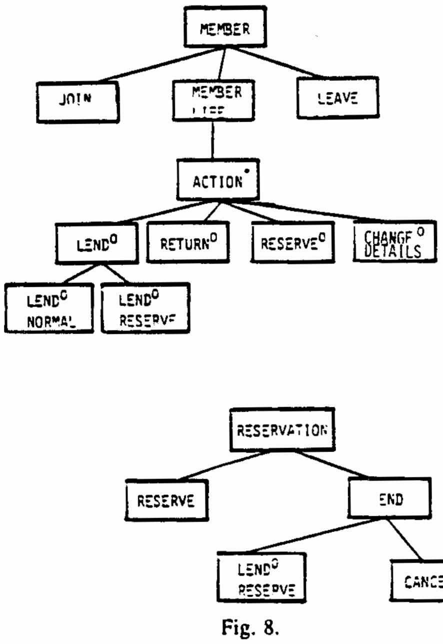


Fig. 8.

constraints imposed by the structures to which it belongs. The library only LEND's a BOOK to a MEMBER if the member has JOINED but not yet LEFT and if the BOOK has just been CLASSIFIED or RETURNED. Looking ahead to the error handling, a LEND input has to be checked against both relevant BOOK process and the relevant MEMBER process.

There is an implied CSP-like [1] parallel composition between processes (i.e., structures) that have actions in common, at least in as much as the processes describe the library itself.

The problem of data integrity is the problem of ensuring that several data items cannot take mutually inconsistent values. For data items within one process integrity is ensured by the process itself, which defines an appropriate update mechanism. When an action happens that is common to two processes, both must execute to keep in step with the external reality. Common actions therefore ensure integrity among data items that belong to different processes.

Interestingly, many systems deliberately allow their data to reach inconsistent states over controlled periods. For example, we may choose to run the BOOK and MEMBER processes as part of various on-line update transactions and the RESERVATION processes as part of a daily batch run; in this case the LEND-RESERVE in RESERVATION is not synchronized with the same action in BOOK and MEMBER.

Interfacing with existing systems also often leads to processes with common actions being left unsynchronized. For example, the BOOK processes may be part of an existing system to which a MEMBER and RESERVATION subsystem has to be added. The existing system already has a means of collecting an input for the LEND action. If the only convenient interface with the existing system is to extract periodically a file of LEND's then the BOOK and MEMBER processes cannot be synchronized.

Obviously, we want to avoid the kind of over-specification that excludes perfectly reasonable implementations. In the JSD specification, therefore, the processes that model BOOK's, MEMBER's, and RESERVATION's are not initially synchronized at their common actions. Synchronization can be added later if it is needed.

Allowing the same action in several processes improves the power of the notation, but still not enough to handle all circumstances conveniently. Suppose that no member is allowed to have more than some number, "lend-limit," of books on loan at one time. Lend-limit is defined from the JOIN and CHANGE DETAILS actions. To describe this constraint we fall back on an informal technique. We introduce a variable to the MEMBER process, set it to zero at the JOIN, increment it at a LEND, and decrement it at a RETURN (that is rigorous enough). Then we informally note the constraint ($N \leq$ lend-limit) beside the structure. We are here describing a library in which exceeding the limit is impossible, not just undesirable. So we are happy for subsequently defined error checking routines to reject a LEND if it would break the limit. If not, the constraint should not be added to the model. In practice this kind of constraint is only sensible if a member cannot take the book without the check being run.

Now consider the following description, in which the relevant actions/events are underlined.

"Film stars often marry but their marriages always end in divorce. They are frequently hired to work on a film but they are always fired for breaking one or other of the terms of their contract."

Fig. 9 describes one possible interpretation of this description and illustrates how there can be more than one structure or process for the same entity. There are no ordering constraints between the events in a film star's married life and the events in his or her working life. The two aspects proceed in parallel. We call different structures that share the same identifier, different roles of the entity.

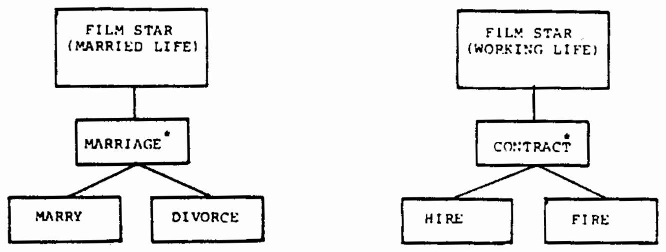


Fig. 9.

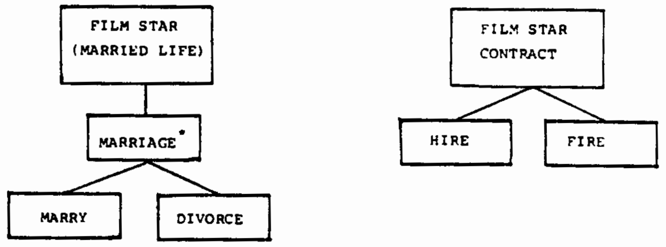


Fig. 10.

Any structure with more than one instance has to have an identifier, or some equivalent means of associating action inputs with their appropriate model instances. For example, a RETURN input has to be directed to the appropriate BOOK process. Thus, every action has to have among its attributes the identifiers of the structures to which it belongs. And if an action does not have a particular attribute, then it cannot belong to that structure; for example, RESERVE cannot be action of BOOK.

Fig. 9 reflects the assumptions that a film star can only have one spouse at a time and one contract at a time. A typical sequence of actions for a single film star is

$$M, H, D, M, D, M, F, H, F, D.$$

Fig. 10 relaxes the second of these restrictions. A typical sequence of actions for a single film star is

$$M, H_1, H_2, F_2, D, M, H_3, F_1, D, F_3$$

where the index on H and F is a contract-id.

More parallelism implies more and smaller structures. Drawing these structures is as much about parallelism as about ordering. A sequence of actions for many film stars is any interleaving of the sequences for individual film stars. (A qualification must be added if film stars can marry each other. MARRY and DIVORCE are then common actions of different instances of the same structure thus placing a constraint on their possible interleavings.)

An entity is defined in terms of its actions. It is simply the suitable name in the top box of a structure, the object on whose instances the ordering constraints apply. BOOK is not defined as a thing with lots of printed pages but as something which is ACQUIRED, CLASSIFIED, LENT, etc. Book would not be a good name if the library also had music cassettes which were ACQUIRED, CLASSIFIED, LENT, and so on according to the same ordering rules as apply to things with printed pages. In practice a developer works with actions and entities together (best of all with phrases like "the member returns the book"). In theory, though, the actions come slightly first.

C. Event Models and Data Models

Database oriented approaches to the development of business systems start by building a data model of the enterprise (interpreting that term widely). Proponents argue, quite rightly, that the users' detailed requirements change very fast; that the stored data from which the requirements are calculated is much more stable; that therefore the key to robust systems is to get the database right; and the way to get the database right is to base it, via the stepping stone of a data model, on a description of the enterprise.

For data-processing systems, JSD can be viewed as a generalization of this approach, a generalization that includes the time dimension in the model of the enterprise. The state of a database (or equivalent files) does reflect the state of the enterprise. However, the way the database changes also reflects the way the enterprise evolves. We argue that it is just as important to capture the dynamics of the enterprise in the description which forms the basis of the system as it is to capture its static properties. In JSD we describe the dynamics first (what happens? in what order?) and then we define the states of the enterprise (the data) in terms of these dynamics (what is stored or remembered about what has happened?). Not only is this more powerful for almost all the problems to which the database approach does apply, but it also extends the applicability of the approach to real-time and other systems in which the data are not of central importance.

In making this generalization, we are also offering some clarification of some of the conceptual difficulties at the heart of data modeling. In his book *Data and Reality* [2], William Kent shows by a series of examples that the terms entity, attribute, and relationship, as commonly used, are very difficult to define and in particular very difficult to distinguish. He also argues that any record oriented description (let alone particular hierarchical, network, or relational descriptions) has important limitations and that these limitations are only difficult to see because our habits of thought are conditioned by available implementation techniques.

Events are the basic medium of JSD modeling, not *n*-tuples of data items. Events have the immediate advantage that they usually directly visible in the enterprise. Many data items are not (TIMEONLOAN, ACQUIREDATE, LOANCT, etc.), even though the data model is supposed to describe the enterprise. The concept of an event is fairly easy to define. The important point, apart from relevance to the system to be built, is that we must be prepared to regard the event as atomic, happening at a single instant of time. We are building a discrete simulation whether or not the enterprise evolves continuously or discretely or both. The choice of actions determines the resolution of our simulation. Actions correspond to the smallest updates that can be made to the database or its equivalent.

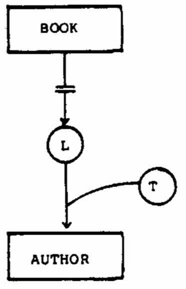


Fig. 11.

An event can have any number of attributes, which simply further describe the event. (For example, in a system to support the use of JSD a single event "AMEND STRUCTURE DIAGRAM OF ENTITY," which was constructed by a whole session with a front-end editor, had the attributes developer-id, time, entity-id, version-id, and the whole of the new structure.)

We have defined the term "entity" as a process (or set of processes sharing the same identifier) that describes constraints on the orderings of the events. Data are defined for an entity by adding variables to its process. The variables can be declared in any way we choose. We can also introduce other processes to hold extra data that do not fit any entity process. For example, we may introduce AUTHOR processes to keep count of the number of LEND's of books by each author in successive periods of time.

(Fig. 11 shows an appropriate fragment of network. The BOOK processes output LEND messages to the appropriate AUTHOR process. The *T* messages define the end of the periods. Author is an attribute of ACQUIRE and not of LEND and so has to be stored within BOOK so the message can be sent to the right destination. These networks are described in Section III. Extra processes that hold data always feed off the basic entity processes in this way.)

Relationships: Relationships are, according to Kent [2], "the very stuff of which data models are made." Although relationships are not a fundamental concept in JSD, they can nevertheless be defined in terms of a JSD model in order to derive a data model of the desired flavor. Deriving a data model is useful if we are heading for a database implementation because most existing techniques of physical database design use some kind of "logical" data model as their starting point.

We would further argue a JSD model clarifies the definition of most relationships. For example, BOOK *X* "is-now-lent-to" MEMBER *Y* if LEND (*X*, *Y*, · · ·) and not yet the corresponding RETURN. The relationship is many-one because in the BOOK structure two LENDs must have an intervening RETURN whereas in the MEMBER structure there is no such constraint. BOOK *X* "[illegible] been-lent-to" MEMBER Y if there has ever been a LEND (X, Y, $\cdots$).

Existence rules are also best expressed by considering the time dimension. The rule "A Y cannot exist unless an X exists" is described by a process in which the CREATE of a Y can only happen between the CREATE and the DELETE of the X.

The following rules will derive a data model from the JSD model.

1) Define relationships between entities where

a) the identifier of one entity is part of the identifier of another (for example, MEMBER to RESERVATION if the identifier of RESERVATION is member.title);

b) the identifier of one entity is part of the state vector of another (for example, MEMBER to BOOK if the borrower is stored in the BOOK process);

c) part of the identifier of an entity is part of the state vector of another (for example, RESERVATION to BOOK where title is the relevant attribute).

2) Normalize any state vectors that are not already n-tuples, adding the obvious relationships between the separated parts. (For example, we could define within the BOOK process a list of all the MEMBERs who had ever borrowed the book. Normalization produces a new data modeling entity but not a new JSD entity.)

The JSD approach breaks down for data and relationships that cannot reasonably be defined in terms of histories of events, for example, for a database describing chemical compounds and their relationships. Except for correcting errors, such data are never changed. Data only need to be changed when something has happened (i.e., an event) that makes the current version inaccurate. The restriction to systems whose databases (or equivalent) evolve is not severe. Still, some static portions of an otherwise evolving database may not be amenable to the JSD approach.

Let us summarize some of the points in Section II. JSD models are defined in terms of events (or, synonymously, actions), their attributes, and a set of processes that describe their time orderings, and by implication possible parallelism. Processes are described by structure diagrams—tree structures whose leaves are the actions. The same action can appear in more than one structure. Usually the description requires multiple instances of a process type, in which case all the actions of a process must share an identifying attribute or attributes. Several processes can share the same identifier; each process is called a role of that entity. Data are defined directly in terms of the model either directly by adding variables to the existing processes or by adding new processes and adding variables to these; in either case data are defined in terms of event histories. The processes, called model processes, are the basis for the specification network.

Actions correspond to the transactions that cause database updates (or their equivalent in a real-time system). Model processes will be converted into database update subroutines. Stored data in the implemented system appears in the specification as the variables of long running processes, mainly of long running model processes.

III. The Network Phase

A. *Elaboration of the Model into a Specification*

The JSD model consists of actions, attributes of actions, and a set of disconnected sequential processes that describe the time orderings of the actions. These sequential processes are the start of the network that will eventually comprise the specification. Development proceeds incrementally, by adding new processes to the network and by elaborating processes that are already there. Three issues must be addressed when a new process is added.

1) How is the new process connected to the rest of the network?

2) What elaboration is necessary to the existing processes to which it is connected? (For example, a new data item may have to be added to a model process, as described above.)

3) The internal workings of the new process must be defined. Unless there is a good reason to the contrary, the internal structure is expressed using the same sequence, selection, and iteration notation used for the model processes.

Processes are added for three main reasons. Data collection and error handling processes fit between the reality and the model. Their purpose is to collect information about the actions and make sure, so far as possible, that only error-free data are passed on to the model. Output processes extract information from the model, perform calculations and summaries, and produce the system outputs. Interactive functions are like output functions, except that instead of producing outputs they feed back into the model. They handle those cases, represented in the extreme by simulations, where the system can create or substitute for what would otherwise have been external events. Oversimplifying somewhat, the model processes hold the main data for the system along with its update rules. The other processes contain the algorithms that calculate and format outputs, and that drive the model either by collecting and checking inputs or by generating new actions.

Some examples of output and interactive functions are described below. We omit examples of input collection processes. For business and for real-time systems, typical examples are on-line data-collection programs and device drivers, respectively. The details of error handling are also omitted. (One technique is to add "guard" processes to filter inputs that look good but which do not fit the current state of the model. Before each read in a model process a write operation is inserted to send a message to its guard describing what the model is prepared to accept next.)

Fig. 12 shows how the four kinds of process in a JSD specification fit together. The network phase can be divided into three parallel subphases corresponding to the three kinds of process added to the network.

In a JSD network diagram the rectangles represent sequential process types and the circles and diamonds the two basic means of process communication, data stream and state vector communication, respectively. Data stream communication is by messages written by one process and

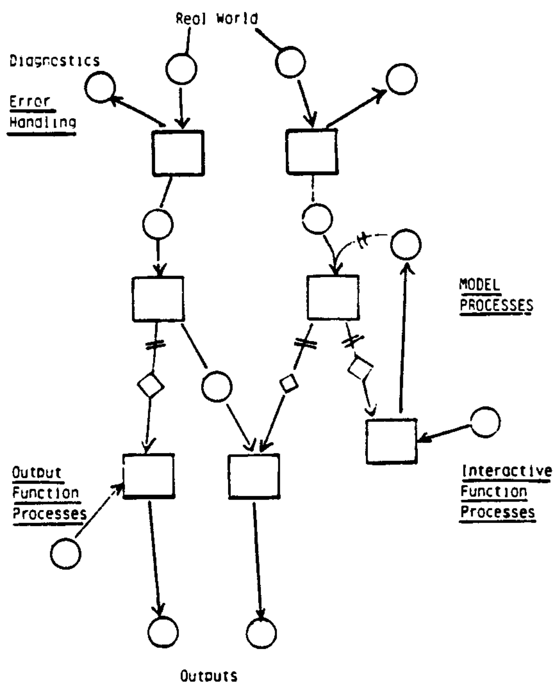


Fig. 12.

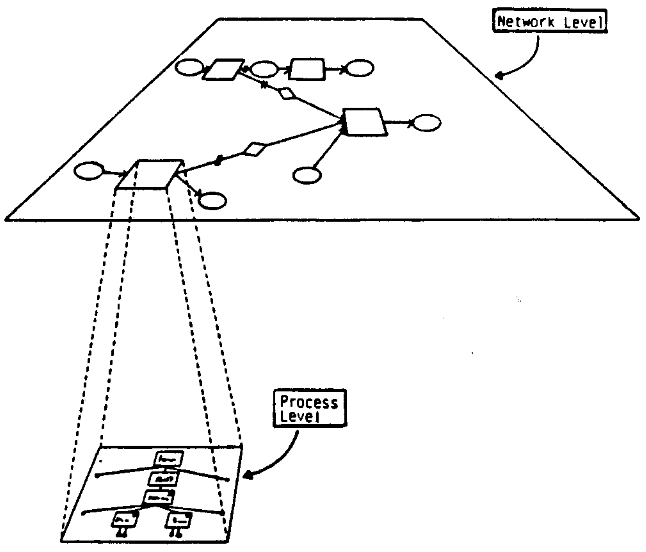


Fig. 13.

read by another. The writes and corresponding reads are not synchronized; the messages can build up in a FIFO queue. In the specification, we assume that the queue is big enough for processes never to be blocked on a write; however, a process is blocked on a read if no message is available. (The BOOK processes spend most of their time blocked waiting for an input.)

The state vector of a process consists of all its local variables including its text pointer. State vector inspection is a form of shared variable communication; one process is allowed read-only access to the other's state vector. We will see below how enquiries about the state of a particular book are answered by a process that examines the state vector of the corresponding BOOK model process.

The double bars in the network diagrams indicate relative multiplicity in the way process instances communicate—the double bars are on the side of the many. (Remember that communication is between process instances, but the rectangles represent process types.) Thus, Fig. 14 indicates that many instances of BOOK write messages to the same NEW BOOKS LISTER and that each process F_j, over its lifetime, examines many instances of the BOOK state vectors.

The merging of the input lines on two or more data streams indicates that the streams are merged before they are read. To the reading process they appear as one stream. The merging algorithm must not starve any stream, must preserve the ordering of messages from one stream, but is otherwise unspecified. This "rough-merging" introduces some indeterminacy into the specification, an indeterminacy that is limited by the kind of overall timing constraints discussed in Section III-C below. Section III-C also contains a discussion of the choice of data streams and state vector inspections as communication primitives.

Fig. 13 summarizes the nature of a JSD specification: there is a network of a large number of sequential processes; each process is, in general, long-running; each process has its own internal structure consisting of sequences, selections, and iterations; the processes communicate by writing and reading messages and by a one-writer-many-readers form of shared variable communication; the state vectors of the processes, particularly the model processes, make up the files, databases, and other storage structures of typical implementations.

The network and the details of the processes are not quite the whole specification. We also need the definition of the actions to describe the way the network is to be embedded into the reality. These definitions fix the specification boundaries for most of the inputs. There must also be an equivalent agreed interpretation for the outputs from the specification. We need to know whether an output circle on the boundary describes a voltage, a [illegible], a screen display, or an invoice printed on [illegible] paper.

We also need information about the desired speed of execution of the processes—this is not part of the specification of the network. Indeed, we are careful to ensure that correct operation of the network does not depend on any particular relative speed of the processes. Nevertheless, the system will be useless if it executes too slowly, and we must complete the specification by describing, if only informally, the required and desired limits on the speed of the processes.

B. Some Examples

First, we will add the following output functions, which can be based on the BOOK model process alone.

1) On input of a given book-id, output the whereabouts of the particular book.

2) On input of a given author, output the titles of the books of that author with counts of the number of copies and the total number of loans for each title.

3) On input of a given title, output whether any books of that title are in the library, and if so, where [illegible].

4) Periodically list the overdue books, grouped by borrower.

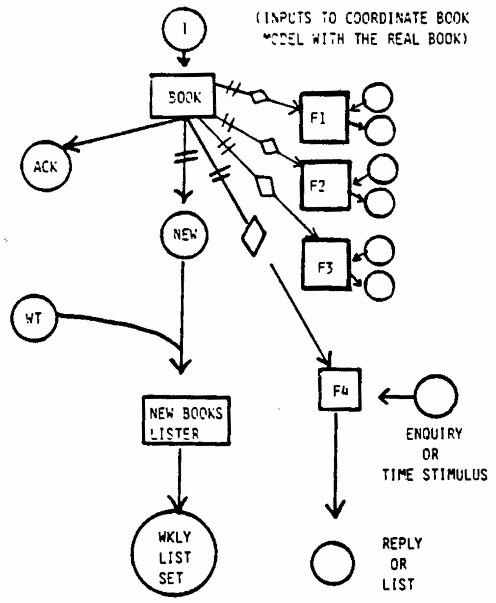


Fig. 14.

These functions are specified by the processes $F1$, $F2$, $F3$, and $F4$ in Fig. 14. Each Fj accesses the state vectors of the BOOK processes. Each Fj uses a particular ideal access path through the state vectors. These access paths are the raw material for the file design part of the implementation step. They are part of the definition of the state vector inspection.

For the above four functions the ideal access paths are as follows.

1) Direct access by book-id.
2) For a given author, grouped by title.
3) For a given title, books with INLIB = "Y".
4) Books grouped by borrower, with ONLOAN = "Y" and LENDDATE LT LIMIT.

In the implementation phase, some kind of file design will be chosen to hold the BOOK state vectors. A sophisticated design will support these access paths directly. A simple design will mean that extra components must be added in front of each Fj to extract from the state vectors actually accessed the ones Fj actually needs. In extreme cases sorting is also needed. The main point is that the network and the details of the Fj's are specified without any commitment to the file design or data storage structures that are to be used.

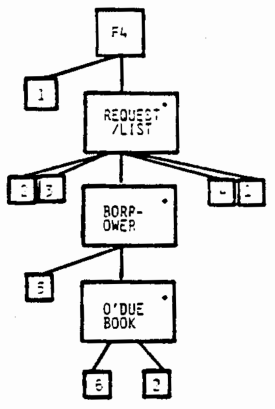
Fig. 15 describes the internal structure of $F4$.

The following two functions require data stream outputs from the model for their formal specification, as shown in Fig. 14.

1) Output an acknowledgment slip when a book is returned.

2) Output a periodic list of new books acquired since the last report. Include the cost of each book, the total value of books acquired in this period, and the brought forward and carried forward totals for the year.

Data streams are used when the model process has the initiative for the communication. The model sends a message to kick the other process into doing something, or at least because it is aware of the messages that need summarizing or further processing to produce the desired output.



```
1.  read next REQUEST
2.  read next BOOK SV   (overdue books only, sorted
                                by borrower)
3.  write LIST-HDR
4.  write LIST-TRLR
5.  write BORROWER-HDR
6.  write  BOOK-LINE
```

Fig. 15.

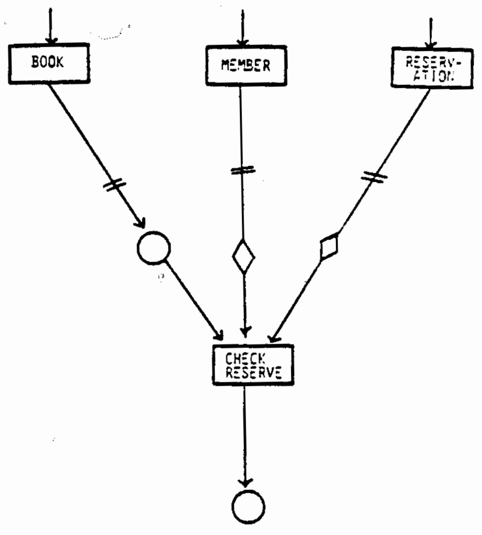


Fig. 16.

State vector inspections are used when the initiative is with the function process. The communication takes place because the Fj's are triggered by enquiries or timing inputs to examine the state of the model.

Figs. 16 and 17 show the network part of the specification for two more complicated functions.

1) When a book is returned check if there are any reservations outstanding for that title and output the name and address of the member who has been waiting longest.

2) On request produce a list of overdue books (grouped by title) whose titles are reserved and also the name and address of the member who has borrowed them.

In the first function the BOOK process sends a message

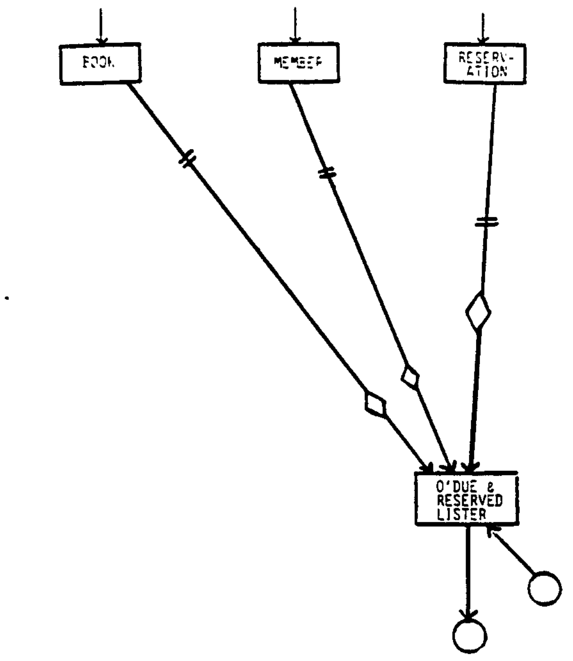


Fig. 17.

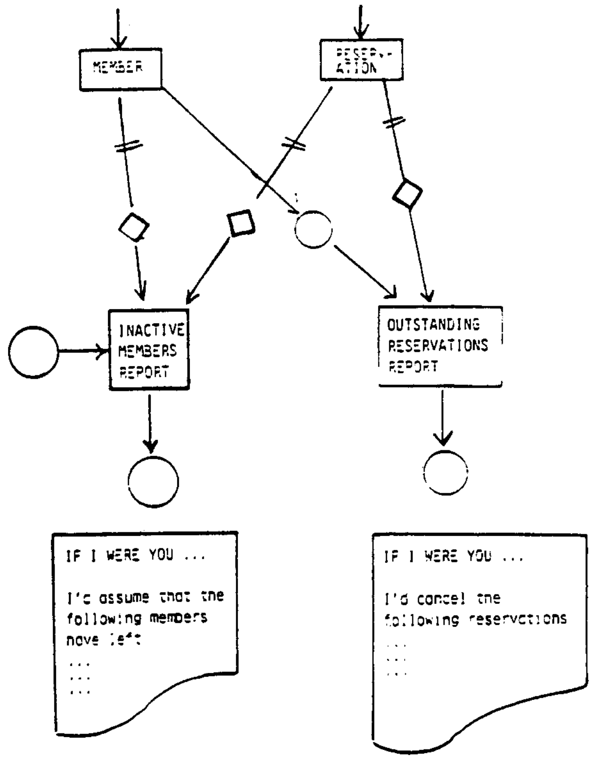

MEMBER
RESERV-ATION
INACTIVE MEMBERS REPORT
OUTSTANDING RESERVATIONS REPORT
IF I WERE YOU ...
I'd assume that the following members have left
IF I WERE YOU ...
I'd cancel the following reservations

Fig. 18.

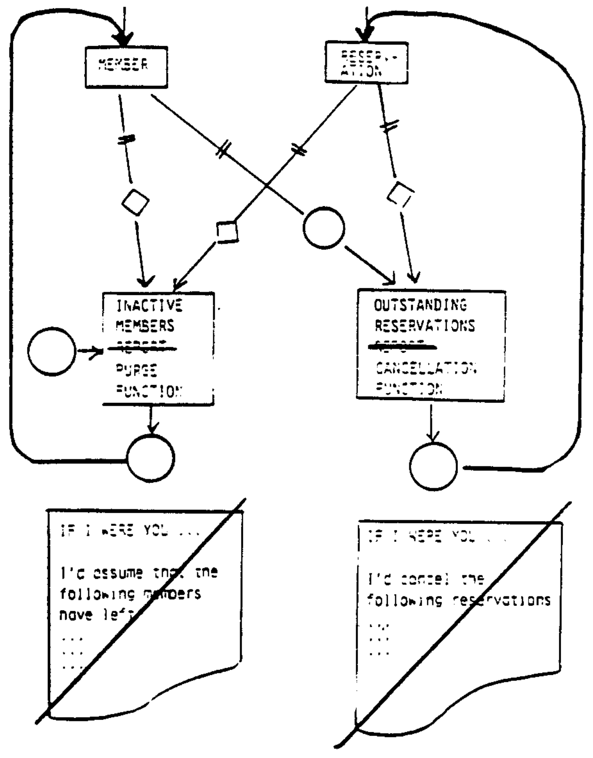


Fig. 19.

to CHECK RESERVE when the book is RETURNed; CHECK RESERVE then examines the state vectors of the RESERVATIONS, and if necessary the MEMBER, in order to output the result. The second function just needs state vector inspections.

The new processes do not communicate directly with each other. We only need to understand their connections with the model. That is why we only need to draw fragments of the network at a time and why each fragment contains some of the model processes.

In each case we have only considered the network part of the specification. We also have to consider the model processes we are connecting to. For each data stream we have to add appropriate write operations; for each state vector inspection we have to check that the data we expect to find has in fact been defined. These details have been omitted, as have the internal structures of the new processes. The JSP programming method [4]–[7] can be used to design these processes.

Fig. 18 shows the network part of the specification for two more functions.

1) When a member leaves, output a list of any outstanding reservations he or she may have.

2) On request output a list of members who have been inactive for at least a year.

Fig. 19 shows a modified specification. The output functions of Fig. 18 have been replaced by similar interactive functions. Now the system automatically cancels reservations when a member leaves, the system automatically makes inactive members leave (the library leaves the member), whether they like it or not.

The system is now generating actions that previously were happening only outside. In a simulation most or all of the actions are generated by interactive functions in this way. The exceptions, for example in a training simulation, are the actions that are still happening externally and for which inputs have to be collected in the usual way. The model processes describe the reality that is to be simulated. The choice of actions and attributes determines the scope and resolution of the simulation. The interactive functions describe the rules by which the actions are deemed to have happened.

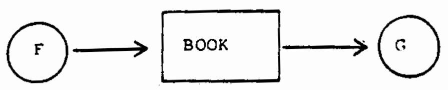


Fig. 20.

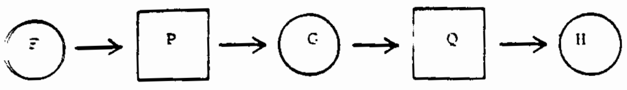


Fig. 21.

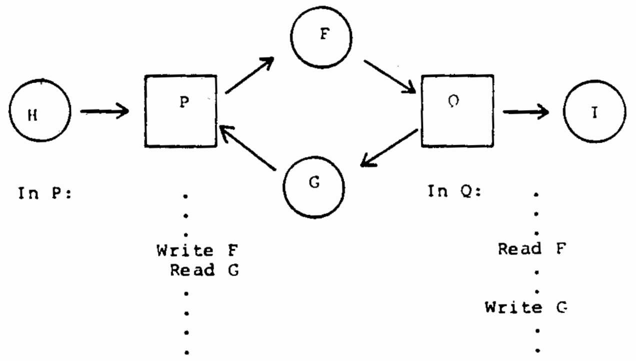


Fig. 22.

Our example also shows that we cannot just look to the external reality to find the initial set of actions.

C. The Choice of Communication Primitives

1) Asynchronous Writing and Reading: The tiny specification in Fig. 20 describes a set of BOOK processes each reading *F*'s and writing *G*'s. The processes are directly executable but the required speed of execution has not yet been specified. Of course the overall speed cannot be very fast—the whole point of these long-running processes is that they can be blocked for months at a read. However, there will be some execution speed of the operations between the reads which is so slow that the results appear on *G* too late to be useful. The specification has to include some description of required speeds. In the present state of JSD such constraints are specified informally by statements such as, "the BOOK's data must not be more than a day out-of-date" and, "any responses on *G* should occur within 5 seconds of the input of the triggering *F* record." Perhaps this informality is a weakness. However, we are reluctant to introduce a more formal notation because there is no means of embodying these timing constraints directly in an implementation in the way that BOOK processes can be converted into subroutines and embodied in the implementation. The extra precision would not really help the implementor.

In the specification in Fig. 21, there would be no point in using synchronous writing and reading on *G*. It would be unnecessarily restrictive. Any acceptable response time between an *F* and an *H* record could be met by making *P* and *Q* very fast and using up the time in the delay between writing and reading. Nor would synchronous communication save us if *P* or *Q* was too slow. The general reason for rejecting synchronous message-passing as our specification primitive is that it often leads to overspecification, that is, to specifications that unnecessarily exclude reasonable implementations.

That is not to say that we will not often choose to implement asynchronous writing and reading by a synchronous message passing mechanism. That is part of our implementation freedom. (In languages that do not support concurrency the easiest way is to implement the messages as parameters passed across a subroutine interface, for example, by introducing into *Q* a suspend-and-resume mechanism such as was described in Section II-A.) It is also part of our implementation freedom to buffer the *G* records for whatever period is consistent with the response constraints. For networks like Fig. 21 and many others, this implementation freedom is worth plenty and costs nothing.

Sometimes we do have to synchronize processes more tightly in the specification. For example, Fig. 22 shows *P* and *Q* more tightly synchronized. Process *Q* is held up at the "read *F*" until *P* reaches the "write *F*." *P* is also held up at the "read *G*" (effectively the same place as the "write *F*") until *Q* not only reaches "read *G*" but also the "write *G*." This is the data stream equivalent of an Ada rendezvous.

2) State Vector Inspection: State vector inspection can be defined in terms of data streams, so it is not strictly necessary to have another communication primitive. However it is very attractive to be able to specify directly inspections of, for example, the data in the BOOK processes. In implemented systems generally there are very many components that have read-only access to stored data, so the extra complication of another primitive seems well worthwhile.

An informal description of state vector inspection uses such phrases as "only coherent states returned," "communication invisible to inspected process" and "a 'get *SV* of *P*' operation returns *P*'s state vector without any operations being executed in *P*." These statements are correct but they skate around the mutual exclusion problem. A particular inspection may be invisible to *P*, but *P* still has to do enough to ensure that the results of inspections correspond only to particular coherent states, for example, to the states of *P* just before the execution of a read operation.

The mutual exclusion can be handled either on the state vector of *P* itself, or on a copy. The two cases correspond to slightly different definitions of state vector inspection. In the first *P* cannot be executing and being inspected at the same time; if *P* is not at one of the approved states the inspection must be delayed. In the second *P* writes a copy of each coherent state; the copy can be inspected while *P* is executing and the result of an inspection may now be out-of-date. Of course, the mutual exclusion problem does not disappear in the second case. An inspection must still be prevented while the copy is being updated.

At first sight surprisingly, we have chosen a definition that fits the second looser description. The reasons are similar to those for choosing asynchronous rather than

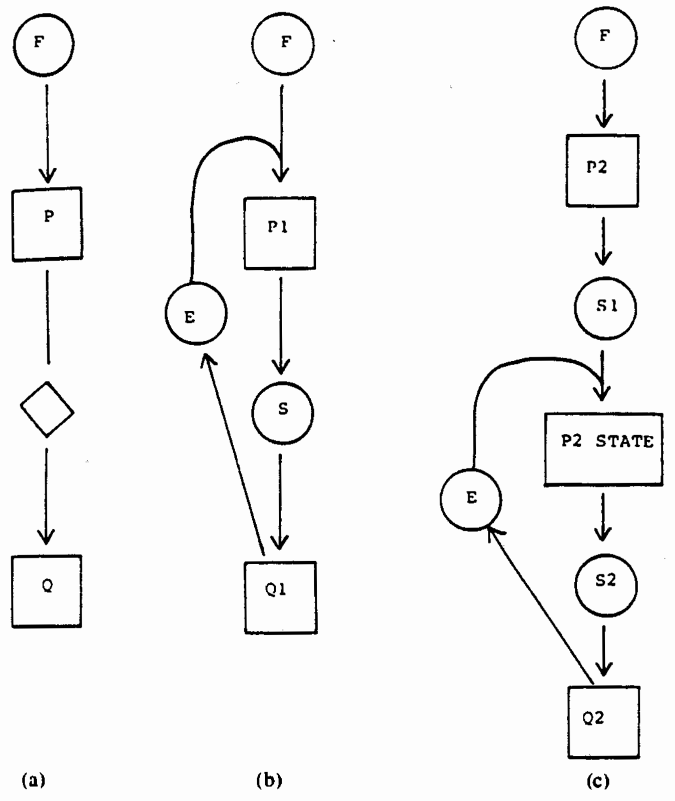


Fig. 23.

synchronous message passing: we gain implementation freedom at little extra cost. Moreover, we are certainly not excluded from implementing the loose definition by using only one physical copy of the data.

The differences between the two possible definitions are clarified by considering their data stream equivalences. Fig. 23(b) is the equivalent of the first, tighter definition, and (c) is the equivalent of our chosen, looser definition. In both $Q1$ and $Q2$ the "get SV of P" operation has been replaced by a "write enquiry on E; read reply from $S(2)$" pair of operations. $P1$ is the process P elaborated to answer the enquiries; it answers by writing a copy of its state on S. The ordinary inputs F and the stream of enquiries E are merged; $P1$ reads the single stream F & E; since $P1$ is a sequential process, it cannot be procesing an E and an F record at the same time.

$P2$ outputs its state on $S1$. P2STATE is very simple; it reads the merged stream E & $S1$; it stores the latest state from an $S1$ record; it answers enquiries from E by outputting this latest state. P2STATE executes concurrently with $P2$. The state stored in P2STATE may lag behind the real state of $P2$.

State vector inspection could be defined by Fig. 23(c) (plus the details of the internals of the processes), or more abstractly along the following lines.

Let $S_1, S_2, \cdots, S_m$ be the chosen coherent states reached during an execution of P. Let $G_1, G_2, \cdots, G_n$ be the "get SV of P" operations executed during an execution of Q. Then the result of each G_i is one of the S_k and:

if G_i results in S_k and G_j results in S_l, then

$$i > j \Rightarrow k \geq l.$$

Let us now return to more practical matters. We have gained the freedom to implement state vector inspection using a second copy of the state vector. This freedom is particularly important in the following cases.

- In distributed implementations, local enquiries are answered by accessing a local copy of the data. The local copy is not necessarily exactly in step with the main copy.
- There is a trend in data processing towards having an operational database which is updated on-line and a decision support database which is periodically updated with an extract from the operational database. (IBM's new database system DB2 may mainly be used for this decision support role.)
- In real-time applications where virtual processors access the same global memory, second copies can reduce the problems of mutual exclusion and interrupt masking.

In our network the BOOK model already lags behind the real BOOK by some indeterminate amount, so there is no extra indeterminacy if the SV copy we access lags behind the real BOOK state vector. Descriptions of overall constraints on processor execution speeds like "for this enquiry the data accessed must be no more than 5 seconds (or 24 hours) behind the reality" limit the sum of the indeterminacies.

With the JSD style of distributed specification, we have to ensure that the problem really is solved in the specification. Because we often fix the relative scheduling of processes in the implementation phase, it is sometimes tempting to build a network that only "works" given particular relative processor speeds, in other words to anticipate the implementation. Certainly, it is unacceptable if some of the processes in the network execute too slowly; that would violate the extra, informally expressed timing constraints. But if a given set of processor speeds does meet all the constraints, then we should not be able to produce unacceptable results by making some of the processes run faster. The (small) price to pay for the implementation freedom we gain by using asynchronous message passing and the looser form of state vector inspection is occasionally some extra care in constructing the specification network.

IV. The Implementation Phase

There are two main issues in the implementation phase: how to run the processes that comprise the specification, and how to store the data that they contain. The first turns out to be particularly concerned with the data streams in the specification, the second with the state vector inspections.

Of course, there may be no work at all in the implementation phase. We only need to find a machine that will execute our executable network of processes. If we can, and if we meet the timing constraints that is fine. We are not looking for extra work.

The problem, though, is the number of (instances of) processes in our specifications and the length of time it takes for their input to accumulate. In our library books last for up to 20 years and we have over 100 000 of them. Will our operating systems and concurrent languages allow us to run 100 000 processes concurrently for 20 years? We often have to combine and package the specification processes into a more familiar arrangement of "short-running" jobs and transaction-handling modules. Admittedly, our library example is fairly extreme. Many real-time environments do support some number of long-running processes, but probably still not as many as in our specifications.

A basic technique has already been introduced in Section II-A. A process can be converted into a subroutine by inserting a suspend-and-resume mechanism at its read statements and by passing the input records as parameters of the call. Every time the subroutine is called it executes part of the long-running program. It is passed an input record and returns control when it is ready to read another.

(The coding details of this suspend-and-resume mechanism can be found in [3]-[7]. The technique can be generalized to allow suspend points at the reads and writes on several or all of the data streams in a program.)

A. Data Design

After we have converted processes into subroutines, we can also easily separate the data (that is the state vector) from the subroutine text so that many instances of a process can be implemented by one copy of the subroutine and many copies of the data. For example, the state vector can be made a parameter of the subroutine. When the subroutine is called it is passed the input record and a state vector; this allows it temporarily to assume the identity of a particular instance; the subroutine executes and passes back the updated state vector when it returns; the calling program is responsible for storing and retrieving the state vectors. Alternatively, the extra parameter may only be the instance-id; the subroutine itself accesses the state vector by retrieval from a file or perhaps by using the id as a pointer into an array.

Once the data are separated, questions of storage and access can be considered. We deal with them here only briefly, not because they are unimportant, but because there is nothing new or special about the way physical data design is handled in JSD. From the specification we have a definition of the state vectors, that is of the data to be stored, and from the state vector inspections we have the definition of how the data are to be accessed. If we wanted we could also map the JSD model on to a data model of any desired flavor. We know the desired response times, the likely volumes, and we can add security and backup information as required. These are the essential inputs from the specification into database design or into the design of storage structures in main memory.

B. Combining Processes

First we show how an abstract network of programs can be combined into a main program and a hierarchy of subroutines. Then we apply the technique to the library example.

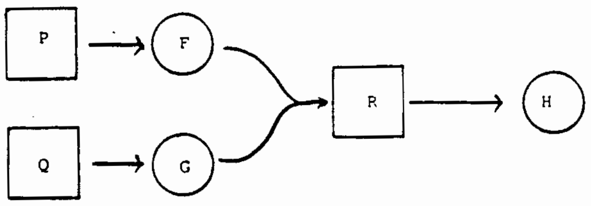


Fig. 24.

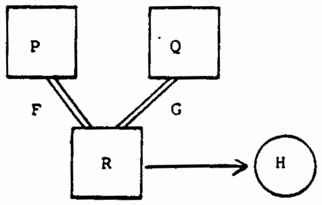


Fig. 25.

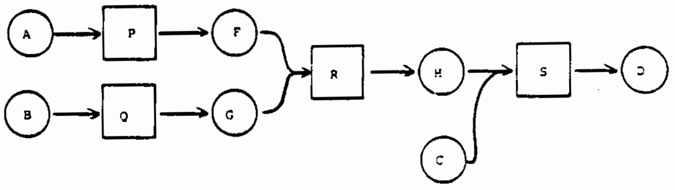


Fig. 26.

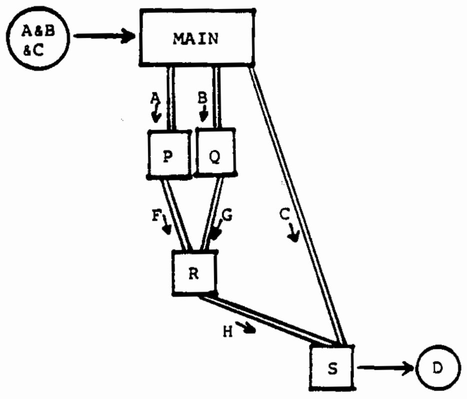


Fig. 27.

Two or more rough merged data streams can be implemented by making the reading process a common subroutine of the several writing processes. In Fig. 24 *P* writes *F*, *Q* writes *G*, *R* reads the merged *F&G* as one stream. Fig. 25 is a subroutine hierarchy diagram in which *P* calls *R* passing *F* records as parameters, and in which *Q* calls *R* passing *G* records as parameters.

Now we will combine the four processes *P*, *Q*, *R*, and *S* in Fig. 26 so that they run as subroutines of a single main program. The technique works as follows. Imagine taking a knitting needle and threading it through all the external input streams, in this case A, B, and C. Pick up the needle and hold it horizontally. The programs will hang in the correct subroutine hierarchy, in this case as shown in Fig. 27. Each program is called by the supplier(s) of its input; all the programs return control upwards when they want to read a new input; the MAIN program is very sim-

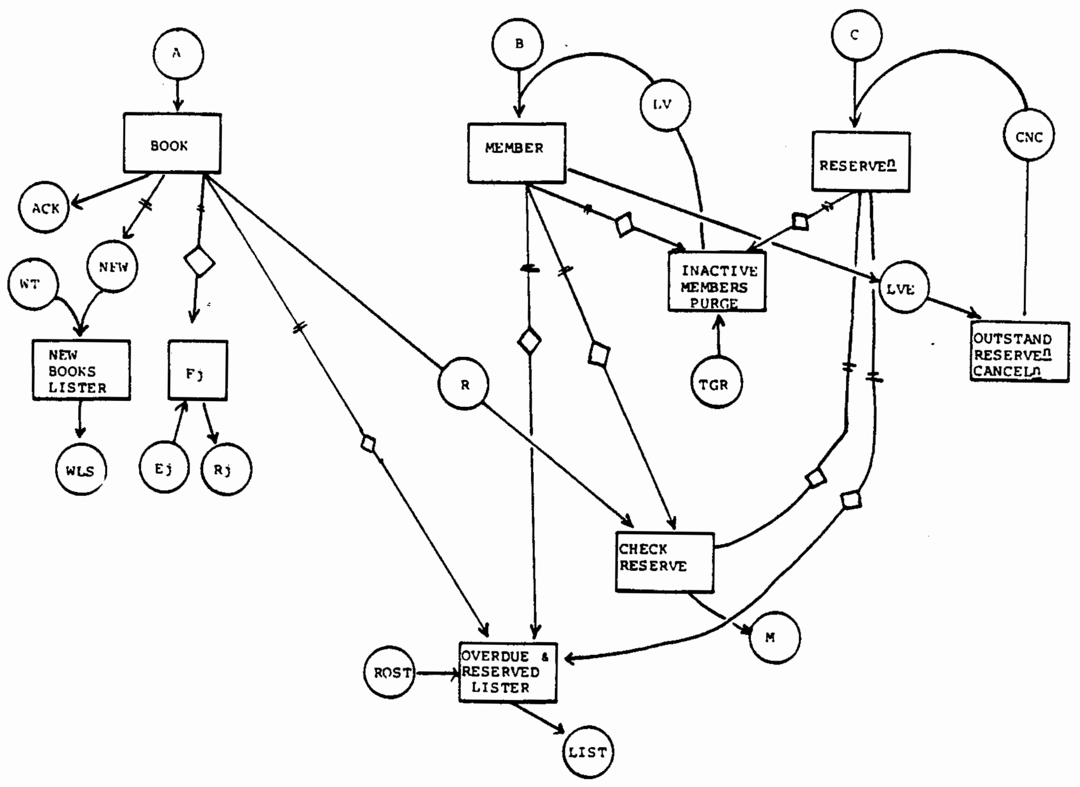


Fig. 28.

ple—it reads A&B&C and calls P, Q, and S with, respectively, the A, B, and C records as parameters.

This technique works provided every program only reads one input stream (the one stream may be the result of merging several) and provided there are no loops in the network. Loops are dealt with below. The treatment of programs with several input streams is omitted: the common special cases are easy; otherwise a more complex suspend-and-resume mechanism is needed.

State vector inspections in the network can simply be ignored. The subroutines will naturally access only a coherent version because the processes only give up control at read or write operations.

Fig. 28 shows the network for the library example. (We did not have to consider the whole of the network during the specification.) Fig. 29 shows the same network rearranged and with the state vector inspections removed. Fig. 30 shows the whole network implemented as a hierarchy of subroutines. A probable internal structure of MAIN is shown in Fig. 31. This style of implementation has no buffering on any of the internal data streams. It corresponds to a transaction oriented implementation in which one input and all its consequences are completely dealt wth before the next. MAIN has only been introduced as part of the implementation. It is a scheduling program; it controls the sequential interleaving of the processes in the network.

C. Internal Buffering

Fig. 32 is the same as Fig. 26, except for the extra data stream E from S to P, which introduces a loop into the network. There has to be some buffering on at least one of the data streams in the loop, on either F, H, or E. By

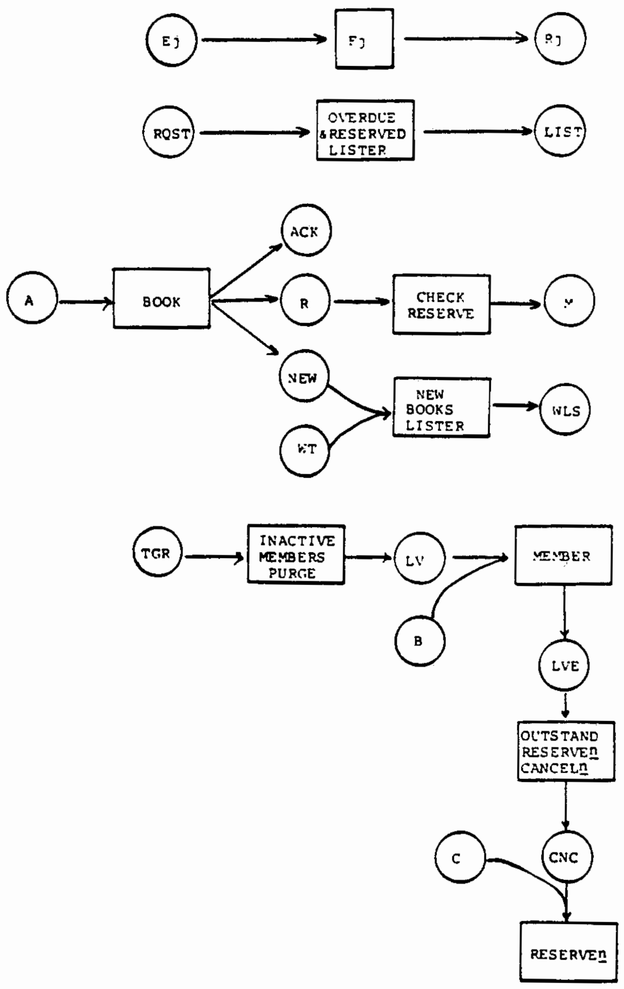


Fig. 29.

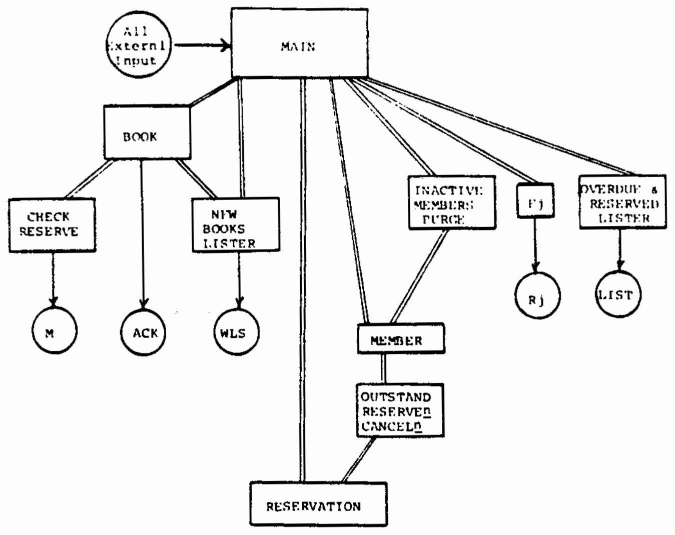


Fig. 30.

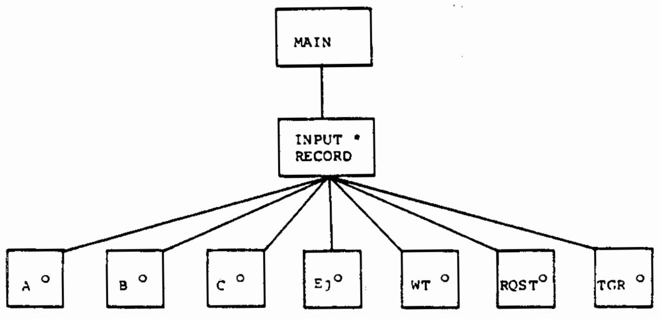


Fig. 31.

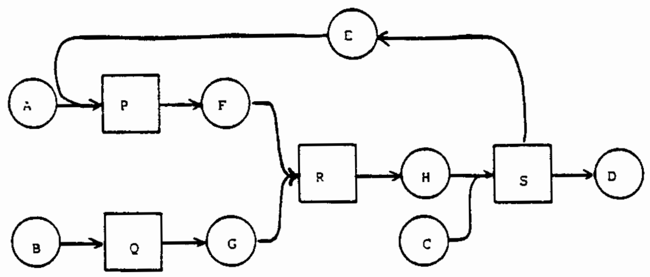


Fig. 32.

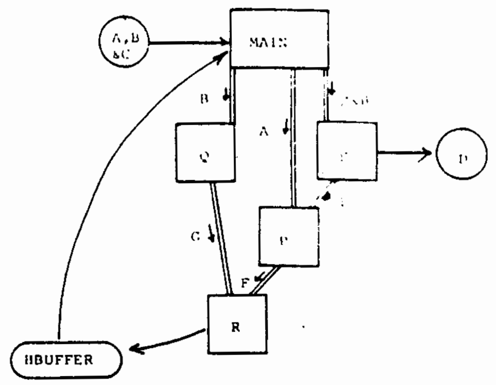


Fig. 33.

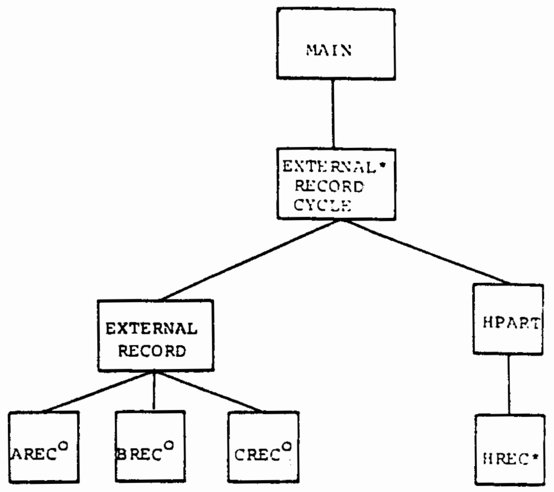


Fig. 34.

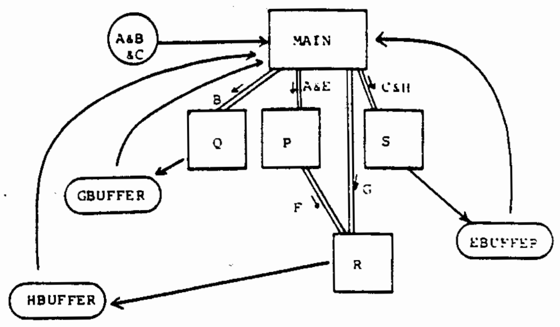


Fig. 35.

cutting the network at *F*, *H*, or *E* and introducing an explicit buffer, the same knitting needle technique can be used to combine the processes into one. The buffer is written by one of the subroutines and read back through the main program. Cutting at *E* would leave the hierarchy the same as in Fig. 27. *S* would write an EBUFFER which is read back through MAIN. Fig. 33 shows the hierarchy with the cut made at *H*. HBUFFER is shown as an oval because it is not a true data stream; MAIN can examine HBUFFER without reading from it.

MAIN now has to make some scheduling decisions; whether to favor *A&B&C* or HBUFFER. Fig. 34 shows a MAIN that empties HBUFFER after every *A*, *B*, or *C* record.

Buffering is necessary to deal with loops, but it can optionally and often quite reasonably be introduced for any internal data stream. In general, the more the buffering the more work the main program has to do. Fig. 35 describes an implementation of the network in Fig. 32 in which *G*, *H*, and *E* have been buffered but not *F*. Notice how different buffering decisions and different algorithms in MAIN lead to different mergings of the pairs of streams *A* and *E*, *F* and *G*, *H* and *C*.

Fig. 36 shows the structure of a typical MAIN program for the kind of batch implementation that makes extensive use of buffering. Many long-running processes have been combined into one long-running program, which can be implemented by a combination of JCL and operator instructions. The bottom line is that if you do not have a long-running computer then you need a long-running operator.

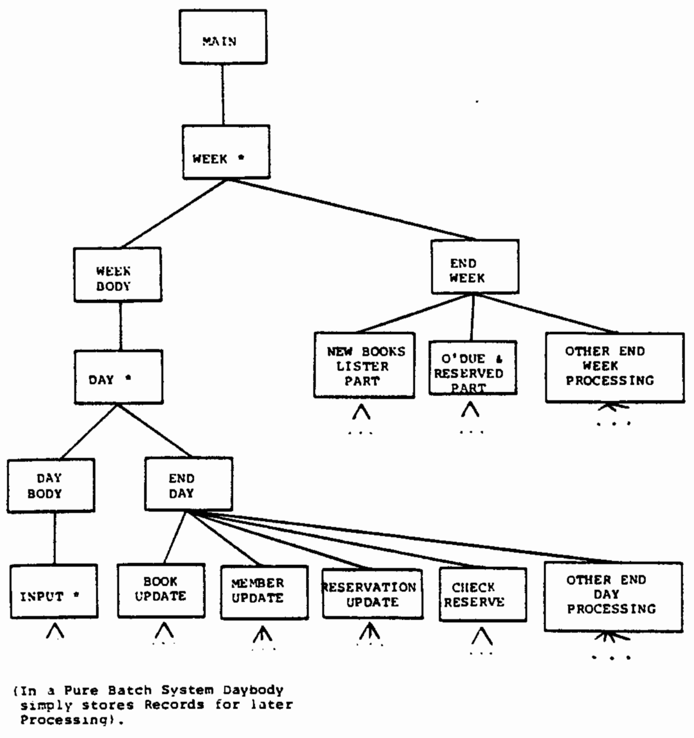


Fig. 36.

D. Implementations with Several Processors

The above techniques combine a network into a single program with or without buffering. Often it is neither necessary nor appropriate to combine all the processes into one. Then the problem is to allocate all the processes to available processors, either real or virtual; to use the above techniques where there is more than one process on the same processor; and to implement the data streams and state vector inspections that pass between the processors.

Between virtual processors communicating via shared memory the state vectors can be put in the shared memory and the data streams implemented by, for example, a circular buffer. Care must be taken in both cases over mutual exclusion. Between real processors data streams will probably be simple messages. State vector inspection can be implemented either by an enquiry and reply pair of messages or by downloading and storing locally a copy of the state vector either periodically or on each update.

Alternatively and preferably we might be able to use directly the facilities of an operating system or of a language such as Ada that supports concurrency.

From among these many implementation possibilities, we have to choose the simplest that meets the informally laid down timing constraints. The plethora of possibilities is not a disadvantage, but a sign of success in separating specification from implementation.

V. Discussion

A. Why Modeling First?

The Meaning of System Outputs: Suppose that the librarian says at an early stage in the development of his system that he would like an output of the form

Number of books in the library = 5921

Average loan period = 12.3 days

What does this output mean? The librarian understands it by reference to the world he knows about. We, as developers, therefore have to understand the world of the library at least well enough to understand what the librarian is asking for. One major purpose of the modeling phase is to establish a basis for understanding and discussing the outputs of the system. That basis consists of the events in the JSD model, their attributes, and their orderings, and it is used not just to define the system outputs but also the data stored by the system and all the terms used in discussions with the users.

If you doubt the importance of establishing this basis, consider what the phrase "in the library" might mean. Is a book to be considered "in the library" if it has been ACQUIRED but not yet CLASSIFIED, or if it has been taken OUT of CIRCulation but not yet DELIVERED? Is the book in the library if it is really out of the library, that is, if it has been LENT but not yet RETURNed? There are many opportunities here for programming the wrong system. A specification document that is signed off, but which lacks a satisfactory model as a basis, is no guarantee of avoiding them.

Comparison to Physics: In any application of mathematics there is a bridge (a mapping) between the reality of the application domain and the formalism of the mathematics. The idea is that the mapping should capture in mathematical form some structure from the application domain. In physics, for example, we might define certain symbols as representing the charge on the electron, the speed of light, and so on; we manipulate the symbols according to some mathematical formalism; we then interpret the results back into the reality via the original definitions. In a similar way the definition of the events is the bridge between the reality of the library and the formal specification (and via the specification to the implementation) of the library system. The manipulations within the specification produce outputs which can be interpreted back into the world of the library via the definitions of events. In this sense building a library system is an application of mathematics to a library.

The bridge between the reality and the formalism can never be completely formal because the reality is not completely formal. Thus, the definitions of the actions are informal dictionary-type definitions whereas the data, for example, can be defined quite formally in terms of the actions.

One of the fundamental ideas in JSD is that the structure of the application domain should be directly reflected in the structure of the specification. The secondary assumption is that for a very wide class of information systems, real-time control and embedded systems and others, the important structure is sequential and should therefore be captured by sequential processes.

Contrast with the Functional View of Specifications: These arguments conflict directly with the widely supported functional view of specifications. According to this, view systems are functions mapping their inputs to their outputs; a specification should define the external behavior of the system, the behavior of the system as if it were a black box; internal structure is dealt with later in a design phase; no internal structure should be part of the specification.

With this view the output "number of books in the library = 5921" is explained in terms of the system inputs. This is, to say the least, uncomfortable for the librarian. Avoiding internal structure is supposed to avoid premature implementations commitments. We argue, however, that the sequential structure of BOOK, MEMBER, and RESERVATION is part of the problem statement and not an artifact of a particular implementation.

Maintenance: The second major purpose of JSD modeling is to clarify the problem of maintenance. The model is defined first and limits the scope of the system. A given model can support a whole family of outputs. The detailed functional requirements change much more quickly than the model on which they are based. Maintenance is easier because we can move relatively easily within the family of functions and, secondly, because we can explain up-front to a user what the family of functions is and therefore what the consequences are of choosing one or another model.

By including ideas of scoping (actually two scopings were discussed in Section II-A) and therefore of families of outputs, we can begin a serious discussion of maintenance. Maintenance is only comprehensible through some concept of families of functional requirements, through some idea of persistence of certain entities and structures through a number of changed specifications. The lack of such concepts is a serious weakness of the functional view. Extreme proponents sometimes argue that there is no such thing as maintenance, that a changed problem is a new problem. However, that is surely bending the facts to fit the theory.

Other Modeling Approaches: At least two other approaches similarly defer consideration of functionality and of outputs. Builders of simulation systems focus first on an abstraction of the reality to be simulated and defer consideration of reports and other outputs.

Database-oriented approaches use data models. We have argued in Section II-C that using events as the basic modeling tool is superior because it extends the modeling to include the time dimension as an integral part of the model and in so doing clarifies the semantics of data and of relationships, both of which are defined in terms of histories of events.

B. Composition and Decomposition

JSD cannot reasonably be called a top-down method (nor for that matter a bottom-up method).

- The network is not built by successive explosions of a process into a subnetwork of processes but middle-out from an initial set of model processes. The complexity of the network is not controlled by hierarchical description but by limitations on the interaction of nonmodel processes.

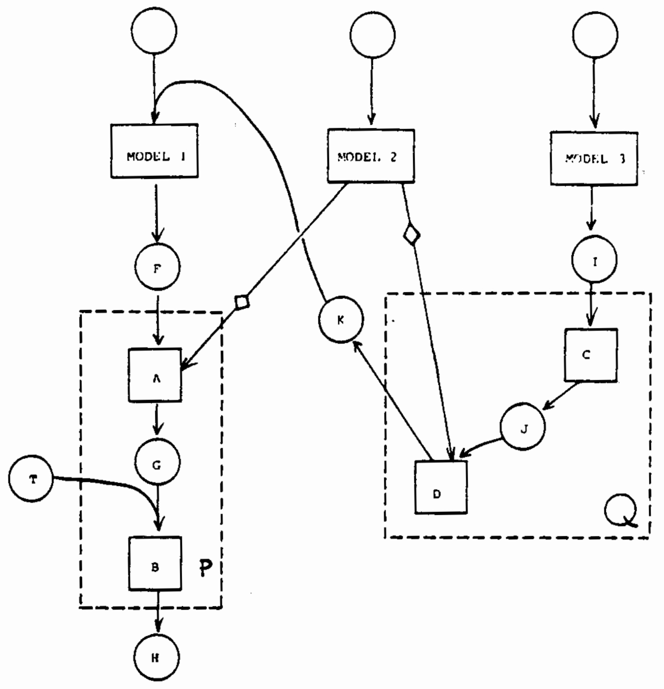


Fig. 37.

- The specification consists of an upper-level network and lower-level tree descriptions of the processes in the network. The whole of the first phase is concerned with the lower level, that is with the definition of the model processes. Then the network is developed middle-out. For each nonmodel process the lower level description is added. Development starts at the lower level of the specification and proceeds up, across, and down.
- Even the development of the tree structures in the modeling phase is not indisputably top-down. Consider the film star example from Section II-B. We started with the actions, the bottom level of the trees. We did not know at the outset whether there should be one top box or two, or even what the top boxes would be.
- Top-down development is characterized by the successive refinement of a single system structure. Using JSD the specification structure is (usually) repackaged into a completely different implementation structure. Fig. 33 and 35 (or 31 and 36) describe two different implementations; each has a different structure from that of the specification from which they were derived; the top levels of each did not even appear in the specification.
- The JSP programming method, used to develop the internals of nonmodel processes, starts with a number of data structure diagrams, composes them into a control structure of the program and then fleshes out the control structure. The data structure diagrams (and also the program structure) are abstractions of the whole program, not a description of its top levels.

Sometimes we do explode a process into a network of processes. Fig. 37 shows the output function process *P* decomposed into processes *A* and *B* and the interactive function process *Q* decomposed into processes *C* and *D*. In the Introduction we stated that nonmodel processes did not usually communicate with each other directly, that they interacted only via the model. That is true of P and Q, but obviously not of A and B or of C and D. The statement is true of the larger-scale structure of the network, that is, if the phrase "nonmodel process" is replaced by "nonmodel process or small subnetwork."

Even here we are just as likely to develop P incrementally by working outwards from the model. We might first add process A which does the basic calculations to produce the output G and only later add process B to do some device dependent formatting of the final output H.

The Weakness of Development by Decomposition: The idea of top-down decomposition (or stepwise refinement) is to develop software by successive decomposition. We start with a single box marked "system." If the system is not trivially simple it is decomposed into parts; the connections between the parts are defined; each part is then considered independently; any part which is not trivially simple is further decomposed.

The motivation is clear: small programs are easier than large programs; large programs are easier than small systems; small systems are easier than large systems. The idea is eminently saleable, especially to management. However, except in a very dilute form the idea is naive. If it were really possible for most developers to make good decompositions, the software problem would have been solved long ago.

The difficulty is this. That first box marked "system" is largely unknown, yet the first decomposition is a commitment to a system structure; the commitment has to be made from a position of ignorance. This dilemma is repeated at successive levels; the designer's ignorance decreases only as the decisions become less critical.

The same argument can be restated as follows. Each decision about the decomposition of a subsystem depends on the decisions that have led to that particular subsystem. This hierarchical decision structure makes the early decisions very critical. A bad early decision may not be discovered until very late. The designer must exercise tremendous foresight to make good decompositions.

We argue that decomposition or stepwise refinement only really works when a designer is effectively writing down a solution he already knows and understands. But then, he is using decomposition as a method of description, not of development. The distinction between the description of something known and thoroughly understood and the development of something largely unknown is often blurred, for example when a developer presents his design to a manager, or when the writer of a textbook presents a solution.

Proponents argue that most software development is on problems that are well understood. This view would be more convincing if most software projects met their deadlines.

If decomposition does not work, or does not work well for problems above a certain size, what is the alternative?

Development by Composition: The alternative is to develop software by composition. Instead of starting with a single box marked "system" we start with a blank piece of paper. Each increment added is precisely defined; an increment is any abstraction of the whole, not necess[illegible] just a component of the whole; part of the system may [illegible] to be restructured, repackaged, or transformed in order [illegible] compose it with another part; at intermediate stages there is a well-defined incomplete system rather than an ill-de-fined complete system. JSD shows that such an appro[illegible] is at least possible.

Finally, while it is a very appealing idea to make mul-tiple descriptions of the same system, in effect to be al-lowed to view the same system from different perspec-tives, the idea does not really become useful unless there is some way to put the different descriptions together.

C. Technical Substance and Managerial Framework

Sensible managers of software projects always produce a plan dividing the project into phases. The manager is concerned with the deliverables at the end of each phase, the user signoff points, the usage of staff, the detection of slippage in time estimates, the political and organizational framework within which the project fits, and other similar issues. The managerial perspective is quite different from that of the technical staff doing the "real" work.

JSD is about the technical substance of a project. The technical substance of a project can be mapped on to a project plan in different ways. This flexibility is necessary because, even among technically similar projects, the managerial and organizational characteristics may vary widely.

For example, since a system may be needed quickly and for competitive or legal reasons, there is no possibility of not going ahead. The main uncertainty is over what and how much can be delivered in the first of several releases. A second system may have a much less certain cost justification, so that throughout the early stages the users only want to be committed to small increments of work, and they want the option of stopping the project as the end of each increment. The project plan, the phasing, and the phase-end deliverables ought not to be the same for these two projects even though they may be technically similar.

Some methodologies concentrate on the management framework rather than on its technical substance. The danger is that the development team gets locked into a framework that is totally inappropriate for their project. That is why many developers do not like methodologies and think them a waste of time. They are forced to produce documents which, in their circumstances, have little value. Their only option is to leave out some steps, which, they are told, means they are not using the methodology properly.

Obviously there are fairly standard mappings of the technical substance of JSD on to project plans. However, flexibility in choosing the mapping includes at least the following:

- considering the most critical implementation issues well before the specification is complete;
- doing a little of each of the model, network, and implementation phases as part of a feasibility or estimating study;

• iterating over the model, network, and implementation phases on a planned series of releases;

• building a low-volume prototype, amending the specification, and reimplementing to produce a high-volume production system.

We can summarize as follows. In JSD specification is strictly separated from implementation and, within the specification, model is strictly separated from the rest. But the ordering of model, network, and implementation is a local, not a global ordering. We need the BOOK model process before the output "Number of books in the library = 5921" can be formally added to the specification, but we do not need MEMBER or RESERVATION. We need part of the model but not all. Similarly, we need some knowledge of the specification to make implementation decisions, but we do not need a complete specification. A project plan fixes the ordering of the work more strictly than is implied by JSD alone; therefore JSD can be mapped in various ways on to a project plan.

D. Tools

Three JSD support tools are currently available, all from Michael Jackson Systems Ltd., 22 Little Portland St., London W1.

• PDF is a graphics editor for tree diagrams and lists of operations (as in Fig. 4) and a code generator from these diagrams into a variety of commonly used languages. The idea of the package is that the diagrams become the source of the program. PDF runs on the IBM PC, VAX VMS with VT100 type terminals and under Unix.

• SPEEDBUILDER is planned to be a series of JSD support products. So far only Unit One has been released. Unit One is a database for holding a JSD specification, a user-friendly editor for the database and a document maker that allows subsets of the database to be printed in a user-defined format. SPEEDBUILDER runs on the IBM PC.

• JSP-Cobol is a Cobol preprocessor that automates the insertion of suspend-and-resume mechanisms like the one described in Sections II-A and II-C, and provides a variety of testing aids. JSP-Cobol runs on a wide variety of mini and mainframe computers.

E. Projects

About 30 JSD or substantially JSD projects have been completed and perhaps as many again are underway. Most, but not all, are data-processing applications. The following are some brief notes about a sample.

• A Fleet Personnel system for a multinational oil company had a ship and employee as its main entities and kept track of people's careers, which ships they were on, where they could join a ship, etc. The system had 120 screen types, about 300 000 lines of procedural Cobol, had interfaces with an existing payroll system, and was implemented under IMS DB/DC.

• A Time Stamp project kept track of employees arriving for and leaving work. They work flexitime around core periods. Different shifts have different core periods, some employees work part-time, some employees are sick or on holiday. The implementation was distributed between IBM Series/1 and System 38 computers. Recovery problems were handled by running the on-line system under a batch scheduler.

• A Fingerprint Checking system restricts entry to a building by scanning in real-time the finger of the person trying to get in. Obviously, response times are critical. Implementation was in Fortran under the operating system RSX-11M.

Projects that are underway include the following:

• the redesign of a substantial part of the on-board software for a torpedo;

• the application software for a communications system to support air defense on the battlefield;

• a set of systems to support the merchandising function of a retail chain, the development of which will take several hundred man-years.

The experience from these projects deserves a more substantial treatment. Suffice here to report that the results have been generally favorable, although for no real project has JSD worked as cleanly or as clearly as it does on the Library example.

Acknowledgment

JSD has been developed (and is developing) within Michael Jackson Systems, Ltd. mainly by M. Jackson and the author, but also with contributions from A. McNeile, J. Kathirasoo, and T. Debling. M. Jackson and I. Smith both suggested a number of improvements to this paper.

References

[1] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, Dec. 1978.
[2] W. Kent, *Data and Reality*. Amsterdam, The Netherlands: North-Holland, 1978.
[3] M. A. Jackson, *System Development*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
[4] ——, *Principles of Program Design*. New York: Academic, 1975.
[5] J. R. Cameron, *JSP and JSD: The Jackson Approach to Software Development*. IEEE Comput. Soc., 1983.
[6] L. Ingevaldsson, *JSP: A Practical Method of Program Design* (in Swedish). Studentlitteratur, 1977; (in English), Chartwell-Bratt, 1979.
[7] H. Jansen, *JSP-Jackson Struktureel Programmeren* (in Dutch). Academic Service, 1984.

John R. Cameron received the M.A. degree and Part III in mathematics from Cambridge University, Cambridge, England, 1973 and 1975, respectively.

From 1975 to 1977, he worked for Scicon, a British software company, mainly on simulations of communication networks. Since 1977 he has worked with Michael Jackson at Michael Jackson Systems Ltd. developing, teaching, consulting in, and building software tools to support the methods described in this tutorial. He is co-developer of the Jackson method of System Development and author of the IEEE-Press tutorial book *JSP & JSD: The Jackson Approach to Software Development*.