Monograph:
Rodriguez-Girones, M.A. and Bennett, S. (1994) Jackson JSD Process, OOP Objects and Implementation of Inheritance in JSD. Research Report. ACSE Research Report 511 . Department of Automatic Control and Systems Engineering

---

Jackson JSD processes, OOP objects and implementation of inheritance in JSD

by

M. A. Rodríguez-Gironés * and S. Bennett *

* Department of Automatic Control and Systems Engineering
University of Sheffield
Mappin Street
Sheffield S1 4DU

Research Report No. 511
March 1994

Abstract

This research report shows how the same software entity can be implemented either as a Jackson System Development (JSD) process or as an Object Oriented Programming (OOP) object, and how a CASE tool that generates the OOP code from its equivalent Jackson process structure diagram can be designed. The tool is designed to be used when a system specification is available in JSD terms and it is desired to implement it by means of OOP languages and techniques; a particular case in which this can be convenient is indicated.

Since CASE tools that convert structure diagrams into code already exist, the shorter approach of transforming process diagrams into object diagrams is also studied and described. It is then shown how the reverse graphical transformation -- from object diagrams to process diagrams -- is equally possible, this permitting systems specified in terms of OOP to be implemented using the JSD methods.

Finally, it is seen how, by successive application of both transformations to its diagrams, the services belonging to a process can be modified (by addition, deletion or change), thus allowing implementation of inheritance and reuse in JSD based software development.

Keywords: Jackson, Structure Diagrams, Jackson System Development, JSD, Object Oriented Programming, OOP, Process, Object, Conversion, Transformation, Inheritance, Reuse.

Table of contents

1. Introduction

In the JSD[1] software development method a system is specified as a set of interconnected, sequential, concurrently running processes, called specification processes. Each process can communicate with other processes in the system, and with the external world, through a series of messages (data streams, in JSD terminology). They can also read the variables of other processes and of external entities (state vector inspections). Each process either models an external entity or cooperates with other processes in providing some system functions. Figure 1.1 shows a System Specification Diagram (SSD), where

- rectangles represent specification processes
- circles represent data streams
- rhombuses represent state vector inspections
- triangles represent rough merges; this means that the reading process will take records from the various input streams in the approximate order in which they arrive.

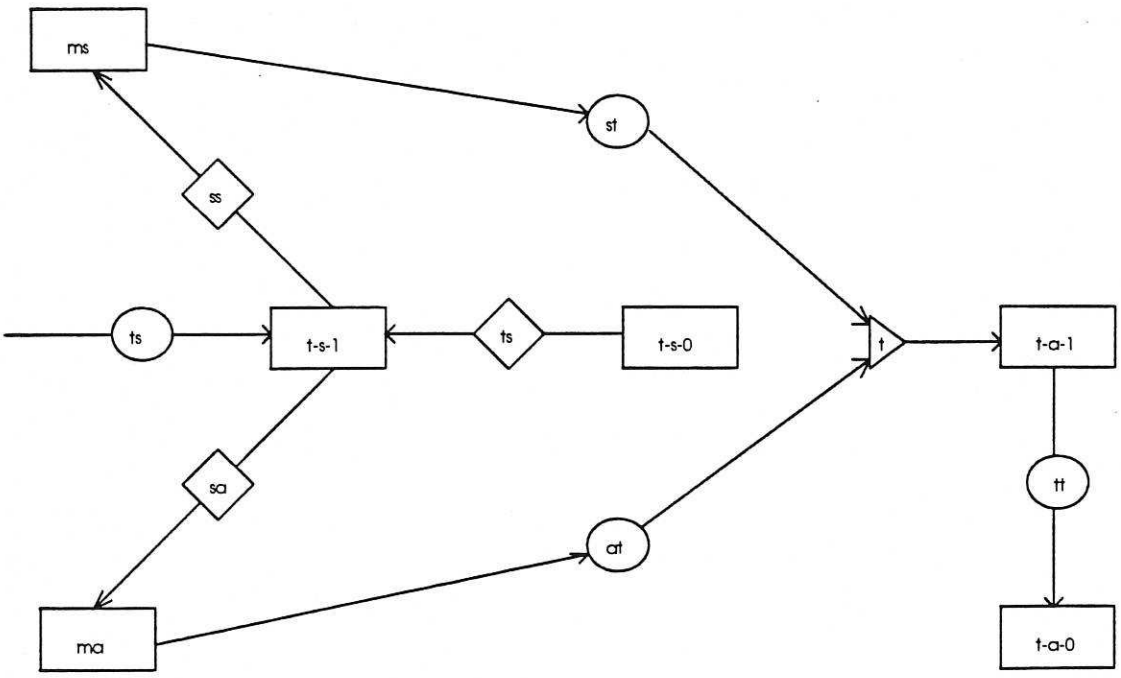


Figure 1.1 -- System Specification Diagram (SSD)

[1]JSD was described in in 1983 in "System Development" (Prentice Hall International), written by Michael A. Jackson on the basis of the work performed by himself and J. R. Cameron. It makes extensive used of programing techniques previously established by Jackson in his "Principles of Program Design" (Academic Press, 1975). Throughout this study we will use the acronysm JSD to refer to either the book or the method.

Processes are specified by means of Jackson structure diagrams, which represent the structure of a process on the basis of the time order of its incoming records; the diagrams also include the actions that the processes perform in reaction to each record. Computer tools are used then to convert the process diagrams into source programs, which will be used to implement the system. Jackson Structure Diagrams can be seen in figures A.1.1 through A.1.3, in the Appendix.

Basically, system implementation takes place by integrating sets of concurrent specification processes into larger sequential implementation processes, which will run as independent tasks. This reduces the number of required processors. Each task includes an intra-task scheduler process (specifically designed for implementation purposes) and several specification processes; the intra-task scheduler reads all of the task inputs and decides which of the task specification processes has to run at any time, on the basis of which of them are required to handle each input message; within the task, the specification processes are inverted with respect their input or output data streams and made subroutines of the intra-task scheduler process or of other specification processes in the task. When the system is implemented on several processors, either real or virtual, or through multitasking, there will be an intra-task scheduler for each of the processors (real or virtual) or tasks. Figure 1.2 shows a System Implementation Diagram where a system has been mapped into a single task; program inversion is represented by a double straight line going from a calling process down to the inverted program. Other transformations, such as state vector separation and process dismembering, also play important roles in JSD system implementation. (See JSD chapter 11, "The implementation step", from where figure 1.2. is taken.)

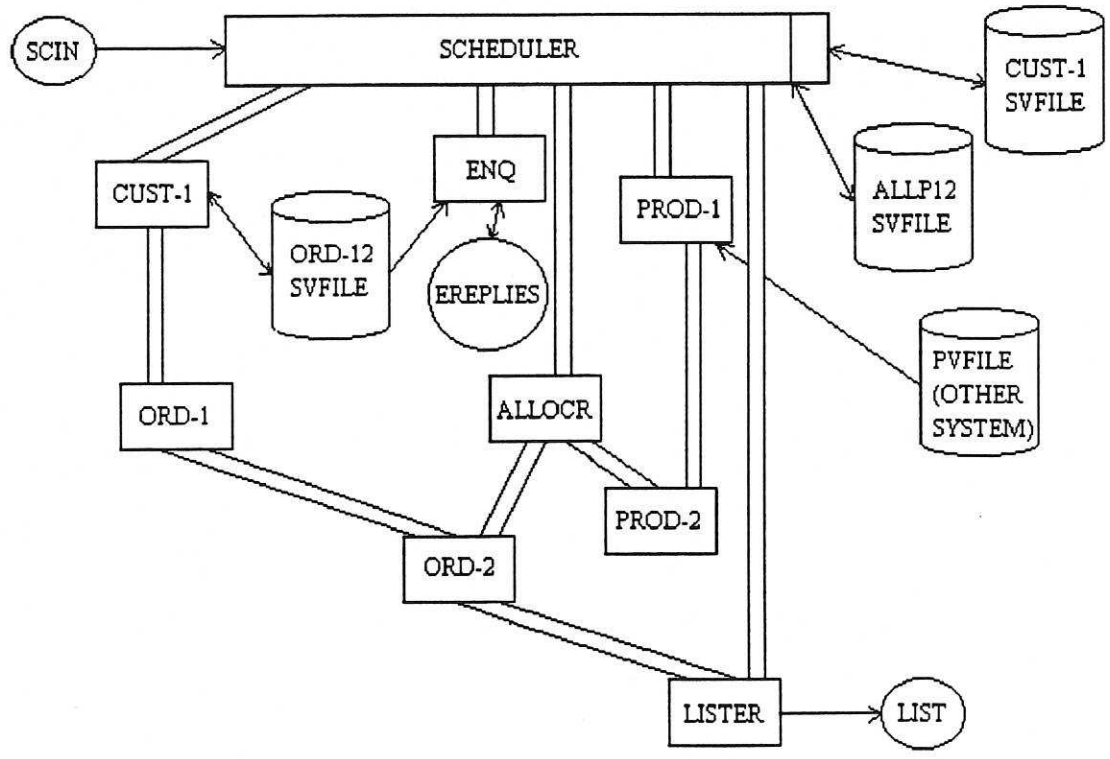


Figure 1.2 -- System Implementation Diagram (SID)

A piece of software implemented as a JSD process can also be implemented as an object, in the meaning given to this term in Object Oriented Programming (OOP), as we will see in section 2.5. And, in certain cases, we may choose, or need, to implement a system specified in JSD by means of objects using OOP techniques and languages, rather than by means of the JSD sequential inverted processes (for instance, when the OOP implementation makes possible a higher degree of parallelism -- see section 2.3).

In such cases, a computer tool which is able to generate OOP source code for the objects directly from the structure diagrams of their related specification processes, would be highly convenient. Section 3 describes how such a tool might work. Throughout the discussion frequent reference is made to a process named process file ABC whose description is in the Appendix "Example 2", and may be worth reading first. (As a sub product of this procedure, the State Transition Diagram (STD) of the process/object can be drawn, and it is shown in section 4.)

Also, taking into account the fact that there are CASE tools that generate code from Jackson structure diagrams, a tool that takes a process structure diagram and produces the diagrams of the equivalent object, which can then be converted into code by one of the existing tools, would be useful. Section 5 describes how to convert process diagrams into object diagrams.

Conversely, it may be convenient, or necessary, to transform a set of OOP objects into JSD processes, and in section 6 we study how to do it in an automatic way. This transformation would allow us to implement certain OOD specified systems with the JSD implementation techniques, though we do not discuss the conditions required in the OOD specified system to make this JSD implementation possible.

By means of the combined application of these two diagram transformations -- from processes to objects and back from objects to processes -- it is possible to automate the transformation of process diagrams to add, modify or delete the process's services. This offers the basis for implementing inheritance in JSD processes and, in turn, it paves the way for effective and intensive reuse in JSD based software development.

2. JSD processes and OOP objects

A piece of program code, which models a real world entity or provides certain functionality, can be structured as a JSD process (a process, from now on in this report) or as an object, in the sense given to this term in OOP (an object). Let us see first what each of these software entities is, how they are similar and in what they differ.

2.1. JSD processes

JSD processes are purely sequential and long-lasting, and they read records (messages sent to them by other processes or by external entities) by means of read operations placed at different points in their program texts. The program text following each of these read operations determines

- which types of message can be accepted at that point
- which actions the process will take upon receiving messages of a certain type, and

- which new read operation in the text will the process execute after finishing this message's processing, or if the process should end at this point.

The last two items will also depend, possibly, on the contents of the message and/or on the current value of the process's internal values. The actions taken by the process may result in changes to its internal variables (its attributes).

All this can also be expressed saying that:

- at each read operation the process is in a different state, identified by the value of its text pointer at that point and further defined by the values of its internal variables
- that only a pre-defined set of messages are valid at each state
- and that the actions taken by the process following each message -- as well as the state in which the process will remain after processing it -- are determined by the current state of the process and by the nature (and may be the contents) of the message.

The state vector of a process consists of its text pointer and its attributes, and we have seen it may change when an incoming message is processed. Otherwise, state vectors can be inspected (read), but not directly modified (written) by other processes; this inspection can take place only when they are in a consistent state, this is, when their owner processes have updated them -- after receiving and processing a message -- and are waiting for another message. [2]

It is important to highlight that:

- each Jackson specification process is a single main one, that must be thought of as running in its own processor
- specification processes do not use (call) and are not used by other processes in the system, but all of them run in parallel and communicate with each other as indicated before
- they are permanently running and, once started, they will continue until they finish; they will only suspend its activity on read operations, to wait for the messages to arrive
- input buffers have infinite capacity, so that no incoming message is lost

---

[2]In System Development, M. A. Jackson indicates that state vector inspection can also take place when the owner process is inspecting another process's state vector or writing to a data stream. But in fact, where parallelism exists, the inspection must take place through an inquiry to the owner process; to avoid simultaneous attempts to inspect and update the state vector, the owner process will accept and service the inquiry only when it has finished processing an incoming message and it is waiting for another, so that its state vector is necessarily consistent. The only fact that the owner is writing or inspecting a state vector does not imply its own state vector's consistency. For instance, after receiving a message, and when it has updated only some of its attributes, it may need to write to another process or to read its state vector; if its own state vector is inspected then, it will be found in an inconsistent state. So, the owner must always be waiting for a new message, stopped at a read operation, when its vector is inspected by means of an inquiry. Another obvious way of maintaining consistency is that the owner writes its state vector to an external buffer every time it completes its update, and other processes read it from the buffer, which takes care of mutual exclusion.

- the segments of code dealing with each specific type of message are scattered around the process text, each one at the point where it must be executed; for instance, process file ABC (figures A.2.1.1 and A.2.1.2) deals with message A by executing A1_actions or A2_actions, depending on its state when message A is received, and A1_actions and A2_actions appear in different places in the process text.

As seen above, specification processes can not be called; they can be sent messages, but they will take the initiative to read them. In this way, we could say they are "active".

As seen in the Introduction, when several processes are implemented on the same processor, they are integrated into larger sequential implementation processes, which will run as tasks in the common processor. Each task includes an intra-task scheduler process and several specification processes; the scheduler reads all of the task inputs and gives control to the task specification process (or processes) required to deal with them; within the task, the specification processes are inverted and become subroutines of the task's scheduler process or of other specification processes in the task. They can be called, as subroutines, by more than one process (for instance, in figure 1.2, which shows one of these implementation tasks, ORD-2 is a subroutine of ORD-1 and of ALLOCR). But, when a specification process has to be integrated in several concurrent tasks, it cannot be called at the same time by two processes from different tasks, as a second call would destroy the return address of the first one (JSD section 11.2, "Channels", page 276); the second call will have to wait until the first one is completed, even if it is made within an execution thread with higher priority.

Figures A.2.1 (made up of A.2.1.1 and A.2.1.2) and A.2.3 show the structure diagram of a process called process file ABC, and its text written in Jackson structured notation; its detailed description can be seen in the appendix.

2.2. OOP objects

An object is implemented as a set of procedures or functions (called operations or services), each of them designed to process only one of the several types of message that the object can be sent. The object also encompasses a set of internal variables (attributes) which are global to all of its operations. There is another object global variable, the state variable, which plays a role parallel to that of the text pointer in Jackson processes: it must be tested by the services, when they receive messages, to check that they are valid at that time and to choose the text to process them. In objects, as in Jackson processes, the state is identified by the state variable and further defined by the attributes. The state variable and the attributes constitute the object's state vector.

Though the state of a software entity is precisely defined by its state vector, throughout the rest of our discussion we will use the term "state" with the meaning "set of states corresponding to a single value of the state variable or text pointer".

The actions to be taken when a message is received -- and the state in which the object will remain when its processing is finished -- are determined by the program text of the service dealing with it, and by the value of the state variable when it is received; the text may also take different paths, depending on circumstances occurring during its execution, which in turn depend on the contents of the message and on the current

values of the object's attributes. The text of the service can then be seen as having the standard structure of a multi-branch fork, with as many branches as states in which the object is ready to process the message and another branch to be executed when the message arrives at any other state, that is, when the message is not valid. This can be seen in figure 2.2.1, where S1 to Sn are the states in which the message is valid, and text_error deals with it when it arrives in any of the remaining states. Figure 2.2.2 shows the structure diagram of this service.

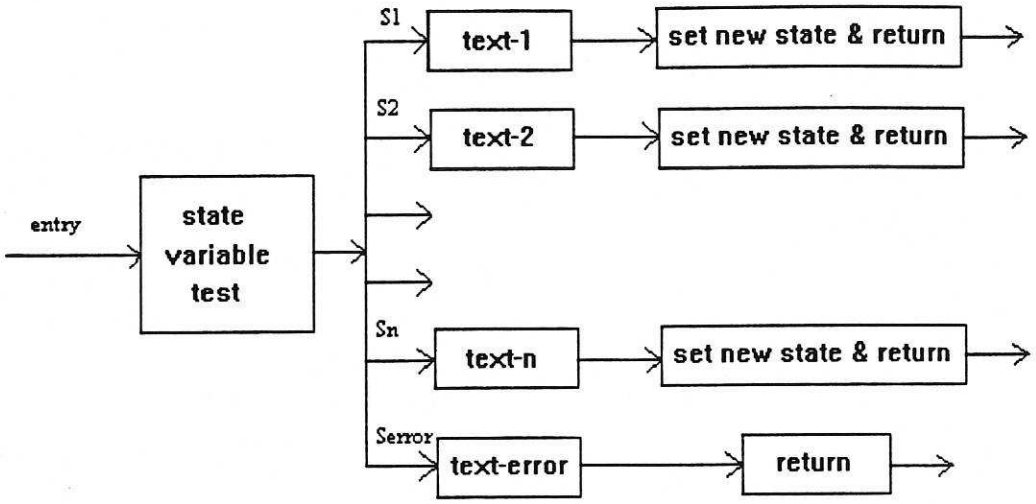


Figure 2.2.1 -- Fork structure of a service

Objects communicate with other objects through message passing. Messages request an invocation of a service provided by an object, the data in the message forms the parameters of the actual invocation. When a service ends processing a message, it returns control to the calling object, and remains stopped, ready to be called again; during its life in the system, a service will remain in this stopped position most of the time.

Objects cannot inspect other object's state vectors by directly reading their internal variables. If they need to know them, they have to call a service whose function is to provide their values.

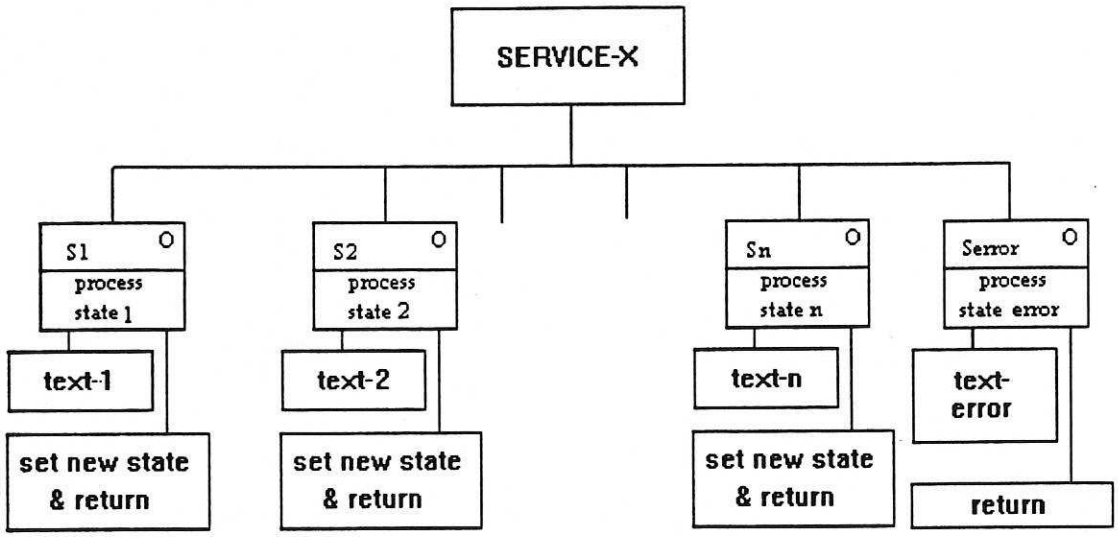


Figure 2.2.2 -- Structure diagram of the service in figure 2.2.1

In contrast with Jackson processes -- where each external or functional entity is implemented by a single long-lasting process, and where the segments of code dealing with each type of message are scattered around that process --, an OOP object consists of a number of procedures or functions which never take the initiative to read their incoming messages, and which run only for the time required to process a message every time they are called. We can see the services as short-running "passive mini-processes". All the code required to deal with each type of message is not scattered around the object, but put together within the same service, the code to be executed at each object state being placed in each of the branches of the procedure or function.

As an example we include in figure A.2.5 the program text of the service message_A of the object implementation of our process file ABC. The object will not issue any read file ABC operations; instead, it will be called by other objects with messages of types A or B or C.

2.3. Possible concurrent execution of an object's services

In certain circumstances, since the services of an object are coded as independent procedures, several of them can be concurrently executed within different parallel execution threads. The required condition is that the subsets of object variables accessed by the different concurrent services do not have common elements (figure 2.3.1) or that, if they do, none of the commonly accessed variables is modified by the concurrent services (figure 2.3.2). In figures 2.3.1 and 2.3.2 the services P and R can belong to two concurrent threads, i and j, but Q can not be entered when P or R are executing (nor vice versa).

Obviously, the state variable can neither be changed, which implies that either the object has a single state, or that none of the concurrent services alters the object's state.

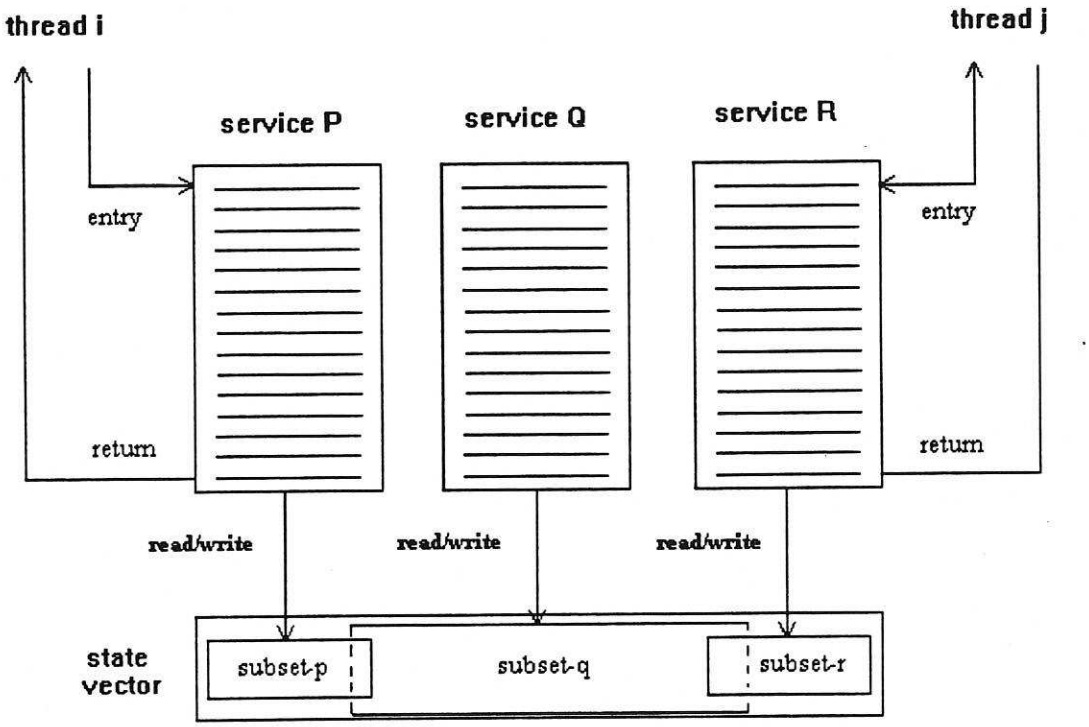


Figure 2.3.1 -- Parallel execution of the services of an object (first case)

This is the reason why, in these circumstances, OOP objects can provide a higher degree of parallelism than Jackson inverted processes. Figure 2.3.3 shows how, when the entities in figures 2.3.1 and 2.3.2 are implemented as Jackson processes, they can belong to only one thread at a time (there are as many possible logical entry points to the inverted process as states, but only one is active at a time, and only one return address is kept).

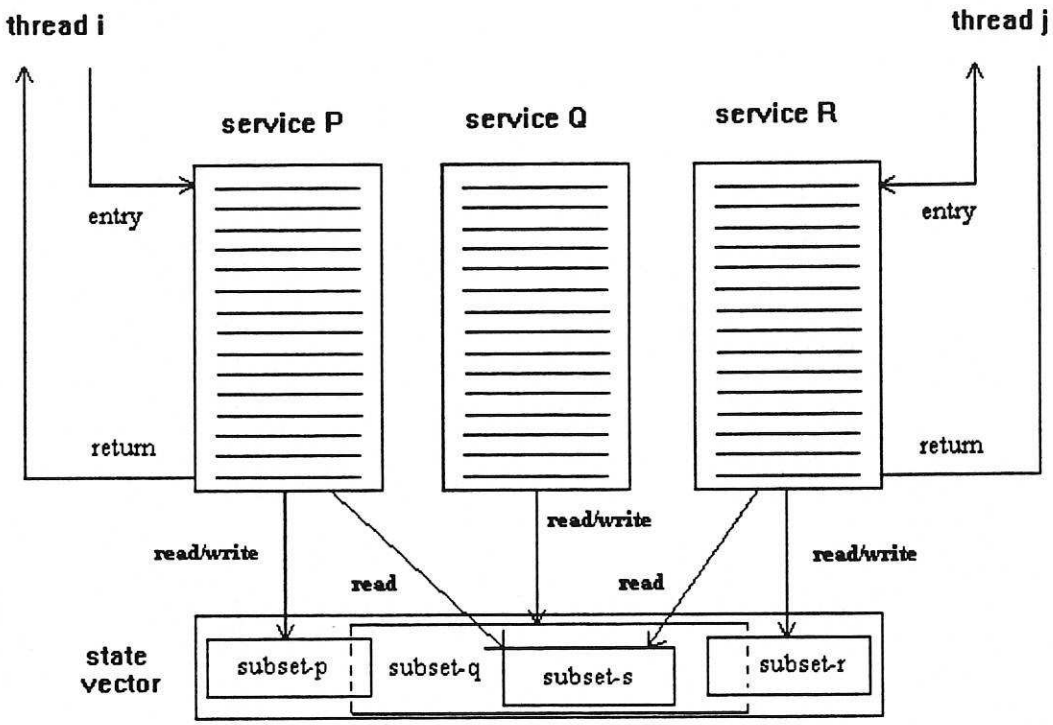


Figure 2.3.2 -- Parallel execution of the services of an object (second case)

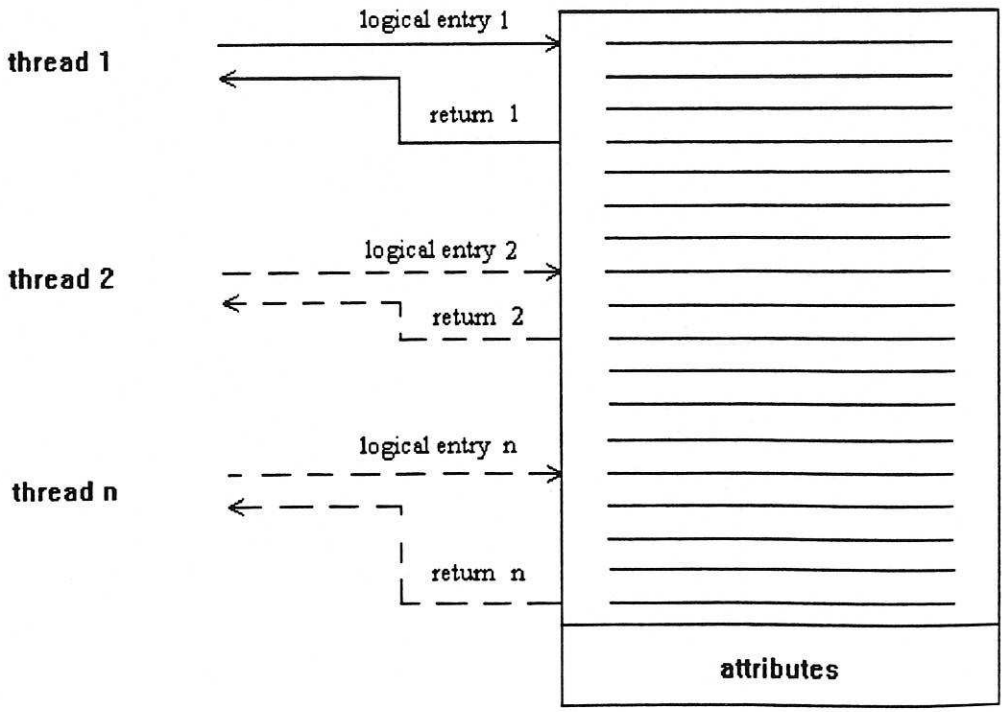


Figure 2.3.3 -- A process can only belong to a thread at a time

2.4. State transition diagrams

Both, JSD processes and OOP objects, can be regarded as implementing state machines, which can be represented by state transition diagrams. These show

- all the states the machine can be in
- the set of message types that can arrive in each state
- the possible actions taken when each type of message arrives (the word "action" includes here "logical actions", like tests and conditional branches required to process the message)
- and the possible states to which the machine may change after each message is processed.

Figure A.2.4 shows the state transition diagram of process file ABC. A state machine is completely defined by its attributes and its state transition diagram.

2.5. Equivalence between processes and objects

A given state machine can be implemented in two different ways, as a process and as an object, and each of them must consist of the same defining elements: state transition diagram and attributes. If we know the process implementation, we may try to derive the object implementation from it.

The different pieces of the text of a service can be found in its equivalent process, and every piece of the process text must map into one of the services. Consequently, an object can be seen as the result of dismembering a process in a number of "mini-processes" (its services or operations) which share the process's variables; they also share a state variable which takes the role played by the process's text pointer.

The branches of the fork structures that form the text of the object's services are, as we have seen, scattered around the process. For each state of the process a number of different messages can arrive. There must be a piece of text in the process to deal with each of these messages; execution of such a piece of text will finish with another read operation representing the next process state. But, for a given state and message, there may be several possible next states, and which one will actually be taken will depend on conditions found while processing the message; the corresponding branch of the service may be, in turn, another fork structure.

If we identify in the process diagram the piece of text (or, which is the same, the sequences of actions and conditional branches) executed for each message at each state, we can write each service program text as a single case structure: it tests the object's state variable and, for each of its possible values, it executes precisely the same piece of text as the process.

The service message_A has been built in this way. Some minor changes to the process text were needed, as figures 2.5.1 and 2.5.2 illustrate

- the read operations ending the different execution segments have been replaced by a sentence which assigns to the state variable the value corresponding to that read operation and a return sentence; this can be seen in service line 1 and service line 2, both of them derived from process line 1
- any Jackson structure (sequence, selection or iteration) starting within the service text and interrupted by a return has been immediately closed after the return

statement, to have a meaningful text, and any text appearing in the process after the read operation corresponding to this return and before the end of the structure has not been included in the service text, for it would never execute; this is the case of the structures process_A_and_B_and_read and read_two_records (which start within the service, at open process structure 2 and open process structure 3, and finish after the return placed at state VII next, at close process structure 2 and close process structure 3) and the group of lines marked as process line set 1, which do not appear in the service text

- closing statements corresponding to structures opened in the process text at a point which precedes the read operation where a segment of the service text starts, have been transformed into informative labels by addition of a colon; process structures 1 through 4 have been changed into service labels 1 through 4, for, in the process, they are opened before the read operations corresponding to states IX and X (marked as process line 2 and process line 3).

```
process_ABC_file:

.....................................................................................
    process_current_key alt(ka=kb)                                  process structure 1
      ka=kb:
      process_A_and_B_and_read seq                                  open process structure 2
        second_process_A:
        A2_actions;
        second_process_B:
        B2_actions;
        read_two_records seq                                        open process structure 3
          read_first_record:
          read(A/B/C);                    --state VII--                 process line 1
          N:=N+1;                                       process line set 1
          read_second_record sel(A read)                process line set 1  open process structure 4
            A_read:                                     process line set 1
            read(B/C);                    --state VIII--  process line set 1
          read_second_record alt(B read)                process line set 1  process structure 4
            B_read:                                     process line set 1
            read(A/C);                    --state IX--    process line set 1
          read_second_record alt(C read)                process line set 1  process structure 4
            C_read:                                     process line set 1
            read(A/B/eof);                --state X--     process line set 1
          read_second_record end                        process line set 1  close process structure 4
          N:=N+1;                                       process line set 1
        read_two_records end                                        close process structure 3
      process_A_and_B_and_read end                                  close process structure 2
    process_current_key alt(ka>kb)
      ka>kb:
      process_only_B_and_read seq
        first_process_B:
        B1_actions;
        read_after_process_B sel(A available)
          A_available:
          read(B/C);                  --state XI--
        read_after_process_B alt(C available)
          C_available:
          read(B/eof);                --state XII--
        read_after_process_B end
        N:=N+1;
      process_only_B_and_read end
    process_current_key end                                         close process structure 1
    take_new_keys sel(A available & B available)

.....................................................................................
```

Figure 2.5.1 -- process file ABC text

As can be seen in figure A.2.5, in the cases where processing of message A was exactly the same for several states (more precisely, for III /IV, for V/VI, and for IX/X) a single common branch has been written. There are other segments of code duplicated, which correspond to actions that are taken in several states within the same service. We will deal with this duplication of code in a further section, and we will try to find how to avoid it.

```
message_A:

.....
      process_current_key alt(ka=kb)
        ka=kb:
        process_A_and_B_and_read seq                          open process structure 2
          second_process_A:
          A2_actions;
          second_process_B:
          B2_actions;
          read_two_records seq                                open process structure 3
            read_first_record:
            status:=VII;                                          service line 1
            return;              --VII next--                     service line 2
          read_two_records end                                close process structure 3
        process_A_and_B_and_read end                          close process structure 2
      process_current_key alt(ka>kb)

        .....

        .....

        .....
message_A alt(status=IX or status=X)
            read_second_record end:                               service label 4
            N:=N+1;
          read_two_records end:                                   service label 3
        process_A_and_B_and_read end:                             service label 2
      process_current_key end:                                    service label 1
      take_new_keys sel(A available & B available)

.....
```

Figure 2.5.2 -- message_A text

3. Generating OOP code from Jackson structure diagrams

The purpose of this section is to describe the internal workings of an automated tool which generates object oriented code from Jackson structure diagrams.

In the first part of this section we will study the basic case of processes reading from a single data stream (or, which is equivalent, performing a rough merge on several incoming streams). Then, in the second part, we will consider some problems related to the time order in which messages arrive at objects, and we will look at processes that read several streams and perform on them a fixed or data merge.

3.1. Basic procedure

The general process to write code for object services, taking as input its equivalent process structure diagram, consists of two logical stages: first the process must be dismembered into pieces corresponding to the different messages and states; then pieces corresponding to a same message must be put together to build up the message's service and, within it, every piece of code must be associated with the state

in which it is executed. Before studying these stages, we will look into the structure of the elementary process execution sequences, that is, the pieces of text executed for a given state and message.

3.1.1. Structure of the process execution sequences

Let us consider a process with I states, named $S(i)$ (where $i=1...I$), which handles J different messages named $M(j)$ (where $j=1...J$). In the process structure diagram, there will be I different boxes containing read operations, which we assume to have been identified and labelled as $S(i)$; $M(j)$ are the names of all the messages, J, appearing in at least one of the read operations, equally identified and numbered. Execution sequences start at the read operations, and can be identified as explained in section 3.1.3. They fit the pattern

$$S(i).M(j).A(ij).CS(ij).C(k_{ij}).A'(ijk_{ij}).S(ijk_{ij}),$$

where

- $S(i)$ is the read operation (defining a State) on which the Message $M(j)$ arrives; none of them, $S(i)$ or $M(j)$, are really part of the execution, we use them here as identifying headers
- $A(ij)$ is the sequence of Actions immediately performed by the process when it is in State $S(i)$ and Message $M(j)$ arrives; it may be followed by $CS(ij)$;
- $CS(ij)$ is a possible Conditional Statement (a computed selection or iteration) found after $A(ij)$ in the processing of the Message $M(j)$ when it arrives in State $S(i)$
- $C(k_{ij})$ (where $k_{ij}=1...K(ij)$) are the $K(ij)$ different possible conditions for the computed selection or iteration $CS(ij)$; in other words, $C(k_{ij})$ are the $K(ij)$ values that the expression controlling the selection or iteration can take (obviously, there is a single value of $K(ij)$ for each pair of ij values; this value is *2* when the Computed Statement is an iteration and, then, $C(1)$ is "*iteration_condition true*" and $C(2)$ is "*iteration_condition false*"); this means that there are $K(ij)$ ijk_{ij} sub sequences springing off each ij root sequence
- $A'(ijk_{ij})$ are the $K(ij)$ sequences of actions that constitute each of the branches where each $C(k_{ij})$ are placed -- in other words, they are the actions performed when each of the $K(ij)$ $C(k_{ij})$ conditions are met;
- $S(ijk_{ij})$ are the $K(ij)$ (possibly different) states which are next to each of the $K(ij)$ $CS(ij).C(k_{ij}).A'(ijk_{ij})$ branches; the read operation in $S(ijk_{ij})$ is the closing statement in the sequence ijk_{ij}.

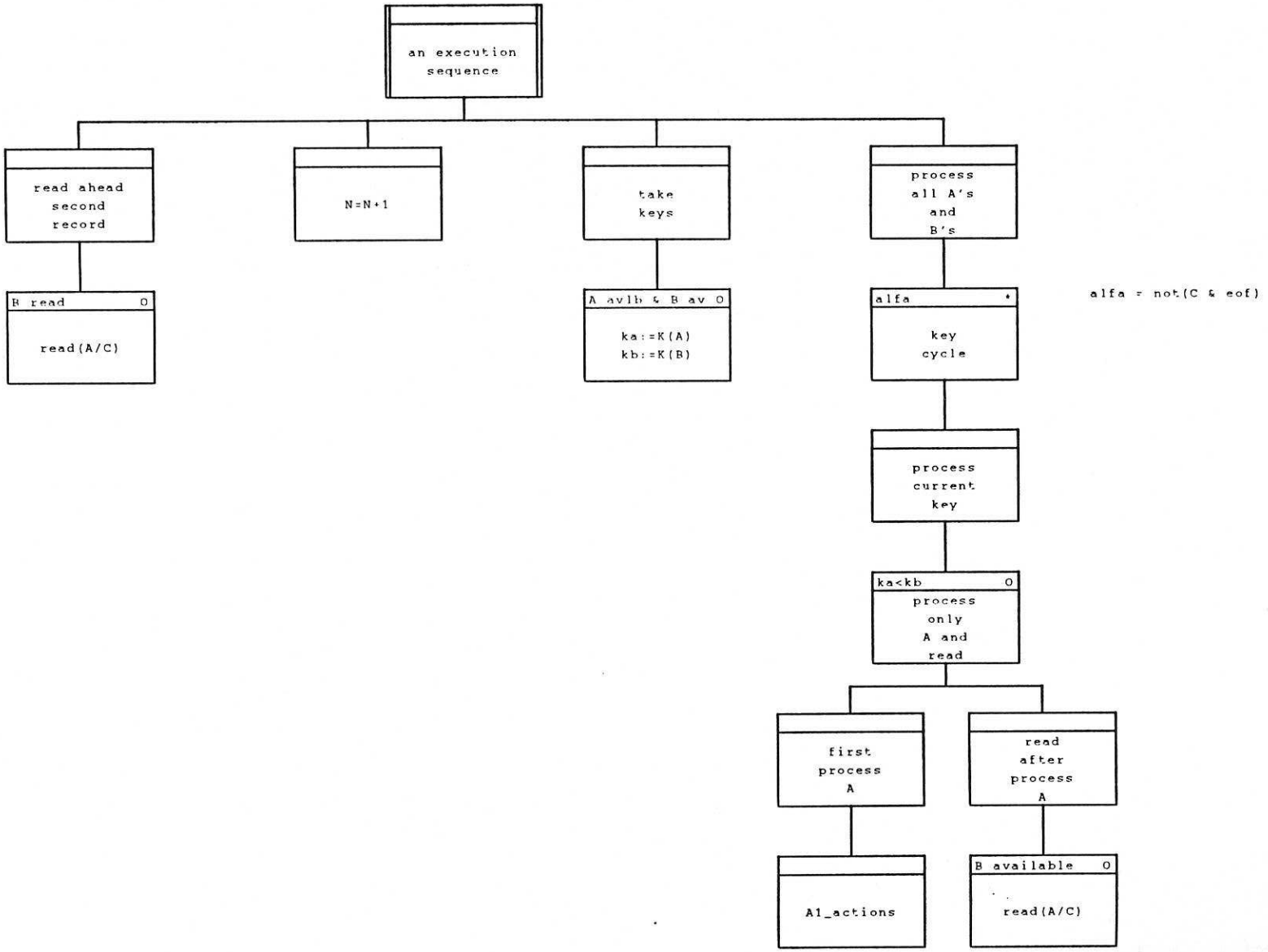
It is clear that some of the sequences ij may not exist.

$A(ij)$, $CS(ij)$, $C(k_{ij})$ and $A'(ijk_{ij})$ may or may not be present in a given sequence. $C(k_{ij})$ and $A'(ijk_{ij})$ only make sense if $CS(ij)$ is present.

Figure 3.1.1.1 shows one of the execution sequences in process file ABC.

SEQUENCE.DAT

Figure 3.1.1.1.



18 - Fi 3.1.1.1 - an execution sequence

3.1.2. Writing the code of an object from its constituent processing sequences

With all the preceding considerations in mind, we can look at the way objects can be derived from structure diagrams by means of an algorithm susceptible of being implemented in an automatic tool.

To write the service corresponding to a given type of message, *M(j)*, we first identify and select all the execution sequences in which *j* appears; we will see in a later section how this can be done. Next, in the language of our choice, we build a multiple branched structure, one branch for each value of *i* present in these selected sequences; in others words, a branch for each of the states in which the message can arrive; this stage is the subject of this section.

There may be more than one execution sequence for a given pair of values *ij* -- this happens when selections or iterations are found in the diagram within the process of the message *M(j)* in the state *S(i)* --, but all the sequences *ij* must start with the same partial sequence -- *A(ij)* --, for it is uniquely defined by the pair *ij*. So if it exists, *A(ij)* must be placed at the beginning of the *i* branch.

Then, if there is more than one sequence for the current pair of *i* and *j* values, due to the existence of a Conditional Statement *CS(ij)*, a language structure implementing *CS(ij)* must be placed after *A(ij)* in the *i* branch; this structure will have an internal branch for each of the possible conditions $C(k_{ij})$, which will consist of the text of $A'(ijk_{ij})$ followed by a statement changing the state to $S(ijk_{ij})$ and a return (these last two operation being equivalent to the read operation in $S(ijk_{ij})$). Only the conditional statement's branches whose corresponding conditions are possible when a message *M(j)* has just been received are to be included in the state *i* branch, for they are the only ones that can be executed in that state.

This procedure, applied to all the *ij* sequences will result in the following service text, expressed in Jackson structured notation:

```
M(j) sel(state= S(1))
        .....
M(j) alt(state=S(p))
        A(pj);
        CS(pj) sel (C(1))
            A'(pj1);
            S=S(pj1);
            return;
        CS(pj) sel (C(2))
            .....
        CS(pj) alt (C(K(ij)))
            A'(pjK(ij));
            S=S(pjK(ij));
            return;
        CS(pj) end
        .....
M(j) alt(state=S(p'))
        A(p'j);
        CS(p'j) iter (while C(1))
            C(1):
            A'(p'j1);
            S=S(p'j1);
            return;
        CS(p'j) end
            C(2):
            A'(p'j2);
            S=S(pj2);
            return;
        .....
M(j) alt(state=S(P))
        .....
M(j) alt(state=others)
        INVALID MESSAGE
        return;
M(j) end
```

where $p=p'=1...P$ are the P values of i for which there is at least a sequence ij or, in other words, the P states in which the message j is valid.

In the text above, the structure for *state=S(p)* has been written assuming that *CS(pj)* is a selection, while *CS(p'j)* has been supposed to be an iteration.

In the branches corresponding to states where there are no *CS* elements, a state assignment sentence and a return (equivalent to the read operation) must be placed after a possible *A* element.

In the external selection there will be as many branches as states in which the message can arrive, and another branch for all the invalid states; this implies that the service disregards any message that arrives at a time when it can not be processed. But messages could be buffered, instead of disregarded, and we will come to this point later.

In fact, the *A'* elements need not be pure actions. They can have a complex structure of the type *Action_ConditionalStatement_Action*. This would simply lead to longer and

more numerous chains, the procedures to apply to each *Action* or *Conditional Statement* element would be the same as the ones applied to *A* and *CS*.

3.1.3. Finding processing sequences in a structure diagram

In the following discussion the order of a structure diagram means the order in which it would be sequentially executed, according to the semantics of Jackson notation. In this context the meaning of preceding element and following element is obvious, and we say that two elements are immediately connected, or that an element is immediately preceding or following other, when there are no other elements between them except, possibly, labels.

To identify all the execution sequences in the process, every identified state *S(i)* must be taken in turn as the first link of a sequence or of a set of sequences. There may be sets of sequences, instead of a single sequence, starting at *S(i)* for two reasons: first, several *M(j)* messages can be valid in that state, each generating a different sequence; then, in the process of each message, conditional elements can be found, in the form of computed selections or iterations, and each possible condition will define a different sequence.

To identify the execution sequences starting at *S(i)*,

$$S(i).M(j).A(ij).CS(ij).C(k_{ij}).A'(ijk_{ij}).S(ijk_{ij}),$$

we inspect the structure diagram in its logical order, starting at the element where the state is *S(i)*. Every diagram element is inspected and incorporated into the sequence, as explained below in more detail. Then we look for the following element in the diagram and operate on it in the same way; if there is more than one following element, the partially built sequence will have to be continued in several ways, one for each of the following elements found; we will call them sub-sequences. This is iterated until a next state $S(ijk_{ij})$ is found for the sequence, or for each sub-sequence. When several sub-sequences are derived from a sequence, because of the presence of a conditional element, we have to take them in turn, one at a time, working on it until it is complete; we must then shift to the next sub-sequence, until all of them are complete.

When all the sequences starting at *S(i)* have been completed, the process of the preceding paragraph is repeated, taking, as new origins for the sequences, each the newly found states $S(ijk_{ij})$. This is now iterated until the diagram is finished and there are no more possible origin states left.

There are well defined algorithms to look for the following elements, as we will see below.

Inspection of the diagram elements requires that the notation used in the chart should unequivocally identify the contents of each box as belonging to one of the types mentioned above -- messages, actions, conditions and labels -- and that this type is recorded in the diagram data base under inspection. For example, a message will be identified by its type name written in upper case as a parameter of a read operation, no actions or labels being written in the same box as the read operation; labels will be written in lower case, and nothing else, except perhaps a process condition, will be written in their boxes; process conditions will be written in a special area, on the top part of the selection and iteration boxes; actions will be represented by numbers, with

their corresponding texts included in a separately accessible list (conditions can be indicated in the same way if needed).

When this identification process is automated, the node information recorded must allow the automated tool to determine the kind of node (sequence, selection or iteration), its parent, its graphical position among its brothers (first, second, etc.), and its children and their relative positions (again: first, second, etc.).

To start the identification of sequences we give the value *S(0)* to the state preceding the execution of the process, represented by a point just on the left of its top box. *S(0)* is taken as the first element of a sequence. This will be the process initialisation sequence; it will not have *M(j)* nor *CS(ij)* links -- as no message has been received and no different conditions can arise while its actions are being performed. The initialisation sequence finishes when the first state *S(1)* is found in the diagram; this sequence will be the only text of the object's initialisation service.

To inspect the diagram sequentially we must be able to identify the element, or elements, following the current one. Precise rules for this have been given by Barry Dwyer and can be found in JSD, section 12.6, page 345. As we must include in the execution sequences the contents of all the nodes in the tree, we need to visit all of them during our inspection, and we have to take into account that:

- nodes that are not leaves, which may only contain labels, precede all their children
- a condition on a box precedes the content of that box.

When building up the sequences starting at *S(i)* we take the box carrying this label as the first element in the sequence, and look for the element following it in the diagram. When we find it, we operate as follows:

- if it is an origin state, *S(i)*, it contains a read operation, and we have to start as many *S(i).M(j)* sequences as *M(j)* messages that are read by the operation; construction of each of them will continue by searching for the following diagram element
- if it contains an action the current sequence or sub-sequence is appended with it, and we search for the following diagram element
- if it contains a selection or an iteration controlled by a computed condition, the current sequence or sub-sequence is appended with it; but now there will be several following elements in the diagram, as many as the possible $C(k_{ij})$ conditions, *K(ij)*, and subsequent exploration of the diagram must follow all those paths, which means that the current sequence may give birth to up to *K(ij)* sub-sequences
- if it is a box with a condition which is part of a conditional statement, and if the condition it shows is possible when a message *M(j)* has just been read and is being processed, the current sequence is appended with it, and it becomes a new sub sequences; otherwise, the box is neglected[3]

---

[3]Obviously, this implies that the machine on which this procedure is implemented can perform logical analysis of the selection and iteration conditions; we do not enter into this problem here. It is also necessary that the conditions are expresed directly in terms of types of records, not in terms of variables related to them. For instance, in the structure diagrams of process file ABC and process files A and B, the condition alfa must be expressed in terms of C and eof, not as a function of ka and kb, if the tool has to find out if it is compatible or not with the records available at that time

- if it is another state, $S(i')=S(ijk_{ij})$ it completes the current sequence or sub-sequence

The first thing to do every time a sequence is completed is to complete any other possible partially built sequences, one at a time. When a sequence is finished and there are no more of them requiring completion, the process is repeated taking as origin a new state $S(i')$, until none of them is left.

3.1.4. Derivation of the code of an object from a process structure diagram; parallel approach

Up to now, and to achieve a clear explanation, we have built up the object operating in two stages, first we identified all the process execution sequences, then we built up each of the services, translating into text and putting into its code all the sequences dealing with the related message, each in its right place.

In practice, this it is not necessary. When a sequence is identified, its text can be directly added to the corresponding service, so that all of them are built in parallel, and the two activities alternate.

3.1.5. Avoiding duplicate code in the object text

Though the code developed using the method explained above is well structured and performs as efficiently as the equivalent process -- for it does not include calls to other functions or procedures --, it includes a lot of duplicated code, and something must be done to avoid this.

In coding the service message_A of the example, figure A.2.5, we have already avoided some duplication by writing a single branch for the states III and IV, another single branch for states V and VI, and also for states IX and X. And yet the three of them contain an identical segment of code, the one comprised between lines *1begin* and *1end* in each branch. The segment between lines *2begin* and *2end*, in branches V/VI and IX/X is also duplicated.

To discuss this subject we will use as an example a process called second process file ABC, that can be seen in figures A.3.1.1 and A.3.1.2, in the appendix. It is similar to the process file ABC that we have used before, but the operations N=N+1, which appear in many places in the latter, have been changed into different operations named M, O, P, Q, etc., and a new one, N, has been added.

We observe that these duplications may have three different causes:

1. A certain piece of code is executed in the process when one of several messages arrives, as it is the case of O in our example. That piece of code will be present in the text of the object services that correspond to those messages. This duplication is due to the structure of objects, not to the method used to get the object's code, and cannot be avoided.
2. There are segments of the process code which are duplicated because of the structure of the process, as it is the case of take keys and take new keys which execute the same function. In principle, they do not need to appear more than once in the text of the same service, and they should be avoided, whenever possible.

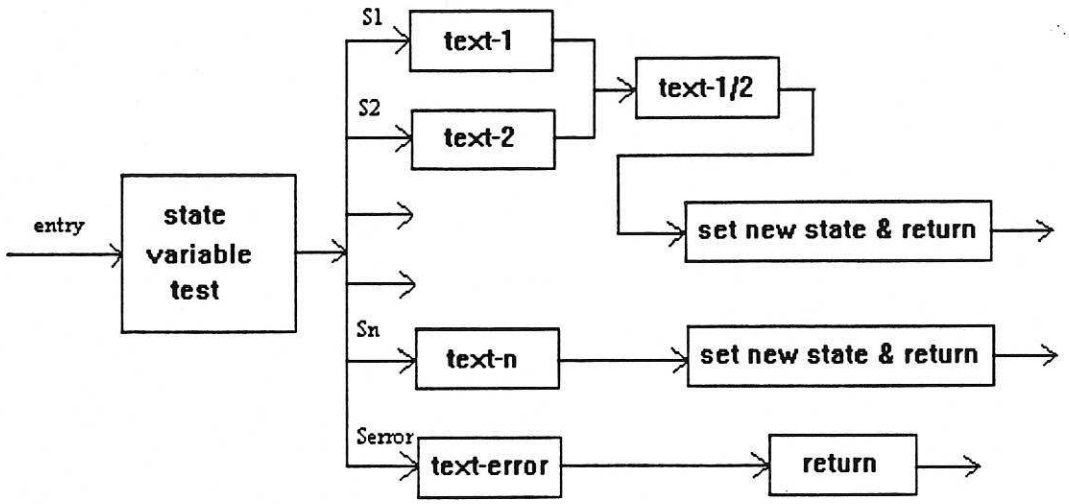


Figure 3.1.5.1 -- Code structure of a service with common trails in its branches

3.a When, in a service, the outer branches that correspond to several different states start in different branches of a same process selection, as it happens with those starting at states V and IX, each of their texts will have an initial part which is private to that service branch, P for V and R for IX, and corresponds to the actions taken within the process selection branch where it starts. This initial part is followed by a second part constituted by the part of the diagram that follows the selection, take new keys in this case, which will be, of course, common to the text of all the branches of this service that start in branches of a same process selection. This means that the general structure of a service, which is a fork, has the particular characteristic that some of the trails of several branches can be common, and the structure can be depicted as in figure 3.1.5.1 better than as in figure 2.2.1, and that its structure diagram has the general form of figure 3.1.5.2, better than that of figure 2.2.2. These duplications should be avoided.

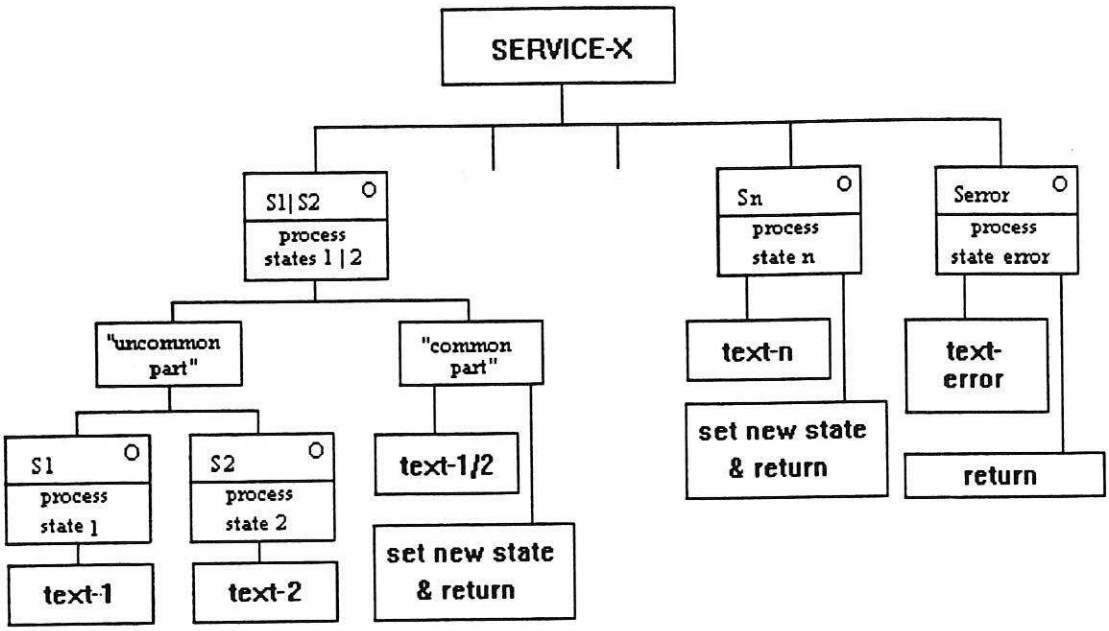


Figure 3.1.5.2 -- Structure diagram of the service in figure 3.1.5.1

3.b Similarly, when an outer service branch starts within an iteration, as for the one starting at state X, its initial part (this is, the code comprised between the beginning of the service branch and the end of the process iteration where it starts, R followed by take new keys, in this case) is followed by the first part of the iteration (since the beginning of process all A's and B's until the set state and return service operations corresponding to each read process operation placed within the iteration). But this first part of the iteration happens to be also the trail of any other service branches starting before the iteration and going into it; for example, it is the trail of the branch starting at state IV. As for the duplication of trails considered in the precedent paragraph, these have also to be avoided.

An observation to make here, relative to paragraphs 1 and 2 above, is that a same software entity may be coded more efficiently -- in terms of code length -- either as a process or as an object, depending on its logical structure. This point may be worth of further study.

It is clear that, in our effort to identify the execution sequences, we have broken the process diagram selections in whose branches there were read operations, and produced as many outer branches of service code as branches had in the process selection; in so doing, we have included duplicated pieces of code at some of the branch trails. We have produced code with the structure of figure 3.1.5.3, instead of the more economical of figure 3.1.5.1.

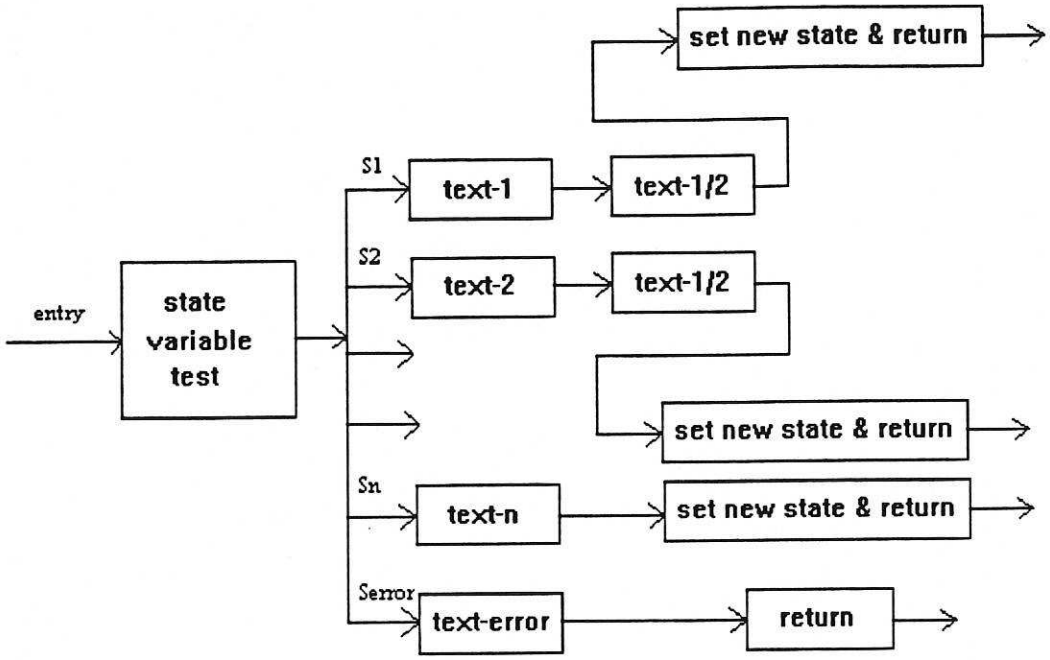


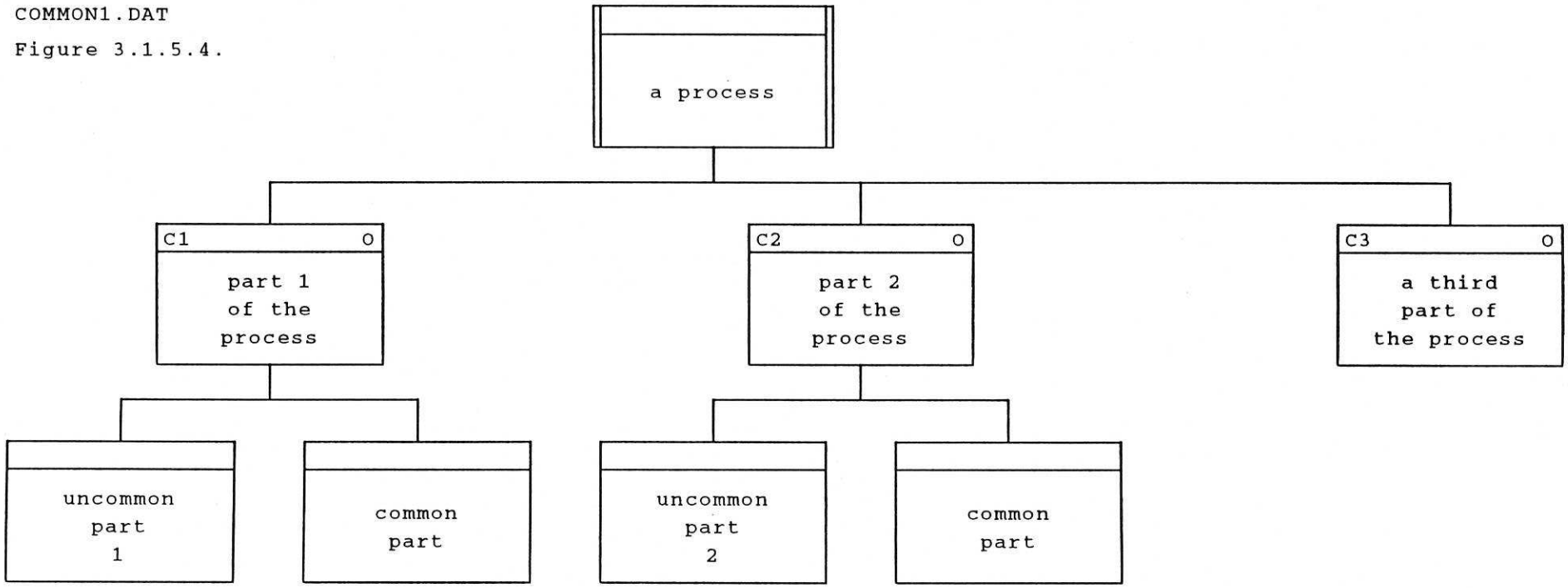
Figure 3.1.5.3 -- Code of the service in figure 3.1.5.1 with the segment text-1/2 duplicated

Also, for each process iteration within which there are state changes, we have produced at several threads of separate code, duplicating the segment corresponding to the beginning of the iteration.

We have to restructure the code to avoid the duplications, and we can do it after observing that, when two branches of a selection of a structure diagram have a common ending part (its trail) preceded by different starting parts, as it is the case of part 1 and part 2 in a process (figure 3.1.5.4), the diagram can be redrawn in an equivalent form shown in figure 3.1.5.5.

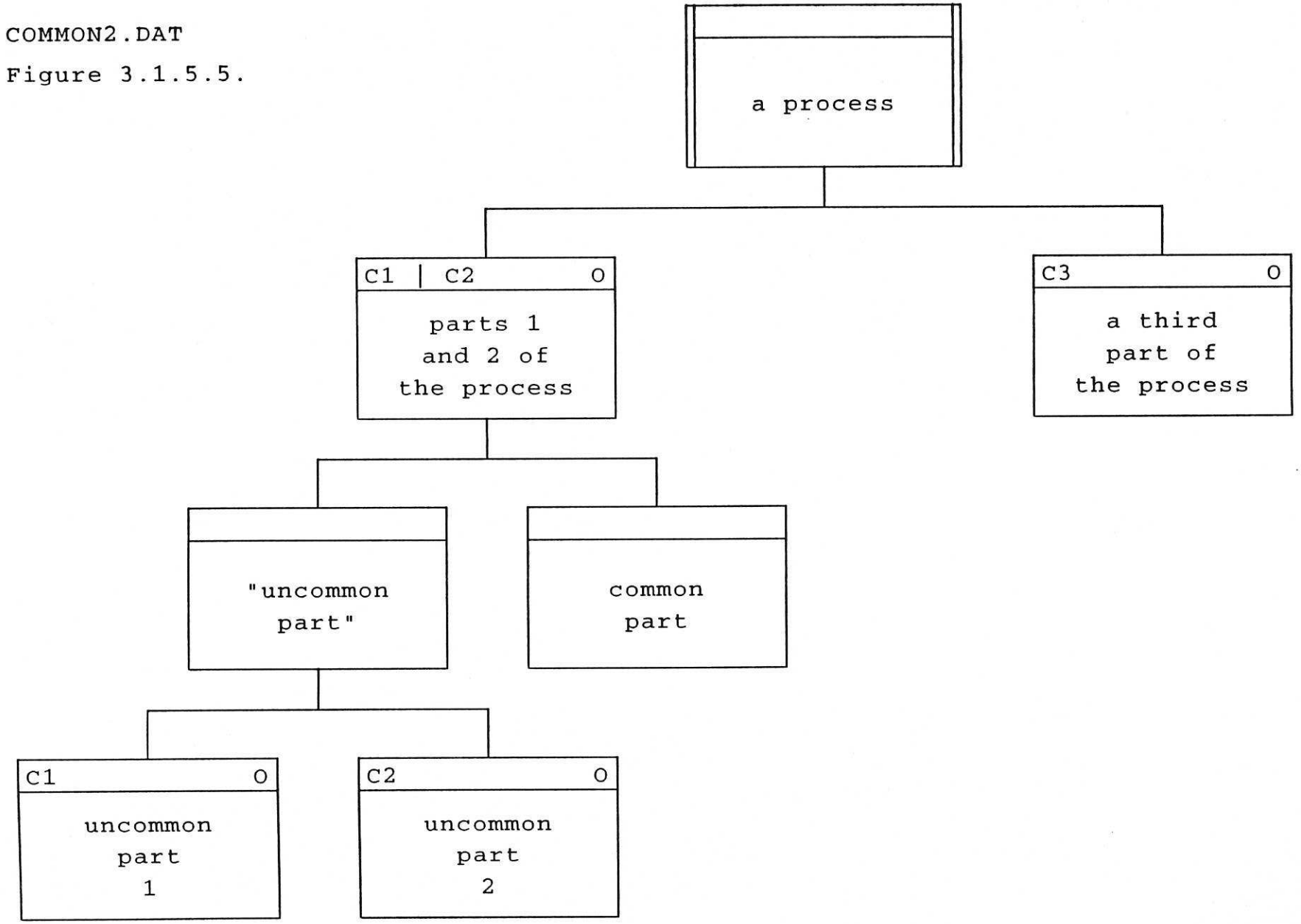
COMMON1.DAT

Figure 3.1.5.4.



COMMON2.DAT

Figure 3.1.5.5.



28 - Fi 3.1.5.5 - a process

This equivalence, expressed in Jackson structured text, is shown in figures 3.1.5.6 and 3.1.5.7.

```
a_process sel(C1)
   part_1_of_the_process seq
      uncommon_part_1
      common_part
   part_1_of_the_process end
a_process alt(C2)
   part_2_of_the_process seq
      uncommon_part_2
      common_part
   part_2_of_the_process end
a_process alt(C3)
   a_third_part_of_the_process
a_process end
```

Figure 3.1.5.6 -- Text of a selection with common trails in its branches

```
a_process sel(C1 | C2)
   parts_1_and_2_of_the_process seq
      uncommon_part sel(C1)
         uncommon_part_1
      uncommon_part alt(C2)
         uncommon_part_2
      uncommon_part end
      common_part
   parts_1_and_2_of_the_process end
a_process alt(C3)
   a_third_part_of_the_process
a_process end
```

Figure 3.1.5.7 -- Text of the selection in figure 3.1.5.6 without duplicated code

So, in order to achieve code without duplications, before generating it, the automatic tool must inspect all the execution sequences to detect all their possible common trails, using the text structure in figure 3.1.5.7 when required.

Finally, duplications imposed by the structure of processes can be avoided:

- first, when two, or more, logically identical sets of elements are found in more than one sequence preceding a common trail -- such as take keys and take new keys, which are preceding the common trail process all A's and B's --, they can be considered to be the same set, and take it as the beginning of the common trail
- second, if the several sets of elements are identical, but they are not placed in that way -- as it is the case of several of the N=N+1 elements in process file ABC -- they can be made a subroutine to be called from the different sequences, as we would do while transforming a process diagram into code.

3.1.6. State vector inspection

Our tool should also generate services to answer to requests for information on the current values of objects's attributes. The requesting messages would be accepted in any state, and their text would be standard. The tool should be instructed on the attributes for which it should include these services.

3.2. Time ordering considerations

We have pointed out the "passive"[4] nature of an object. It never takes the initiative to read messages coming from other objects, its services are just called with the parameters of the messages, and these calls can come from any object at any time. The object can in no way influence the order in which messages are received and processed. When a message arrives, the object can process it, if its state allows for that, or it must disregard it.

Jackson processes, on the contrary, are "active", they read data streams from their input buffers, and they decide in which order to read from them. But when a process reads a single data stream with different kinds of records, or it rough merges several data streams, it must take the records in the order they appear, the same as it happens to an object.

Because of that coincidence of behaviour between object and process in that particular case, it is possible to implement a software entity which reads a single file either as a process or as an object. The conversion of process file ABC into an object, that we have carried out, relies on the assumption that the object will receive a series of A, B and C records that are in exactly the same order (and, of course, are the same) as in the file ABC read by the process.

Nevertheless, it is most likely the object will receive the sub-streams A and B from different objects and, even if the internal order within each of them is the same as in file ABC, the relative order of A's with respect to B's is no longer kept.

Several problems, that we review in the following paragraphs, stem from that lack of synchronism between A and B in the object implementation.

3.2.1. Insufficiently identified input records

The structure of the file ABC clearly determines that, when a first record of type A has been read by process ABC and a second record is read which happens to be C, as, for example, when state=II, it means that there are no more B records in the file, and when C is the second record in a pair whose first one is of type B it means that no more A are available. So, for process file ABC, C can represent the end of file condition for either the sub-file A or for the sub-file B; its meaning is not contained in itself, but depends on its order in the file ABC.

But, when we change a process into an object, it can not know the actual meaning of a C message until the next one is received. For instance, when the object is in state=II, after A has been read, if C arrives, it can mean the end of the B stream, but it may also happen to be the end of the A stream, which arrives before the first B.

This means that a process like process file ABC which processes input files with records whose meaning is not absolute, but depends on their position in a file, can not be transformed into an object; if we do it, the object is flawed, as it may mistake the meaning of C. For the conversion to be possible, all the input records must have a

---

[4]The words "active" and "passive" are used here in a way different from the one they are used at other times, for instance, in HOOD, where they indicate the presence or not of a control thread in the object; in this report they only aim at pointing out the way in which processes and objects receive messages.

precise meaning, independent of their position in their respective files. In our example there should be a record C_a signalling the end of the A stream and another C_b for the end of B, and process file ABC should react to each of them in a different way.

3.2.2. Need for buffering

Leaving apart the problem posed by the insufficiently identified record C, we have also the problem that this conversion of process file ABC into an object assumes that messages will arrive at the object in the same order as the equivalent records have in file AB, which is not necessarily true: many A messages, for example, may arrive at the object before a first B arrives, while it needs to read a B (or, in any case, its eof condition) before it can process an A, and vice versa. The object as implemented disregards any messages arriving when its state does not allow for their processing what, in the case of our example, will probably make the object completely unuseful, although, in other cases, it may be just what is intended.

This situation is parallel to the one we have when a process like process file ABC reads a stream ABC that is the result of rough merging two separate data streams A and B, since they will not be synchronised: If we do not want to lose all the messages which arrive at a state not right for their processing, then, either process file ABC must have buffering capability, or there must be a pre-process, with buffering capability, which reads and rough-merges the files A and B and produces a file ABC with the right order, which will be read by process file ABC. The same has to be done when working with objects, and, when buffering is needed we will be either converting into an object a process with buffering, or converting a process like process file ABC and its rough-merging, buffering, pre-process. In any case, we will carry on the transformation with the techniques seen in the preceding sections, but we do not need to put in the object(s) anything that is not in the process(es); buffering devices will already be present in these, if needed.

Obviously, this need for buffering only arises when the process/object has several possible states. When there is only one possible state, and all types of records/messages can be read at any time, there is no need to buffer any of them.

3.2.3. Processes with multiple data streams

So far we have used process file ABC in order to avoid dealing with processes that read separate data streams. Now we examine processes with multiple data streams and use process files A and B.

Processes of this kind read from one or other of their several input data streams in an order determined by the process itself (may be on the basis of the record data), not in the order they arrive: for example, if the process wants to read a record A and there is none available, it will wait for one to arrive, ignoring any number of B that can be ready to be read. As this is completely different to the behaviour of an object, we have worked on another process able to read a single file ABC, whose records are in the same order in which process files A and B would have read them.

However, the main difference between these two processes is that the one reading separate streams does not need any other buffering than that supplied by the data streams (which are infinite capacity buffers), while the other needs either an internal

buffer or a merging pre-processor with buffering capability. When the second one is transformed into an object, this will need to incorporate that capability as well.

Nevertheless, when a process reading separate streams is converted into an object, we must incorporate the appropriate buffers and buffer manager into it, though they are not present in the process, because messages from the several streams can arrive in an order different of the one in which the object wants to process them. Otherwise, we operate the transformation as explained in the preceding sections.

These buffers and buffer manager can be generated in a standard way, and it must be possible to instruct the automatic tool about which record types need to be buffered. Every message arriving "out of state" will be buffered and, after every message is processed and a new state entered, the buffer manager will check if there is any buffered message that can be processed now, before it returns control to the calling object.

In retrospect, we see that it was not necessary, nor useful, to change the reading operations from process files A and B into those of process file ABC to take into account the possibility that a pair of records A and B starts by B. We could have worked with a process with the same reading structure as process files A and B which read a single file ABC exactly in the same order as this process would read the records from the files A and B; the buffers would take care of any stream or pair AB starting by B instead of by A.

4. Automatic drawing of the process/object state transition diagram

Once all the states and execution sequences have been identified, they can be used to draw the state transition diagram, operating in the following way.

1. To start with, the states must be ordered according to their position in the structure diagram, first those on the left side and, among them, first those on the upper side.
2. A list of states and execution sequences must be made keeping that order; every state must be followed by all the sequences starting with it.
3. A check must be made to see if there are states in the list with exactly the same sequences following them. If there are, they are the same state, even if it is represented by more than one read operation in the structure diagram, and it needs to appear only once in the STD. In figure A.2.4, we can see this is the case for states III, V and IX in process file ABC (and also the case of II, VIII and XI and that of I and VII and of IV and X).
4. Then we draw a box to represent *S(1)* in the sate transition diagram and put its name within it.
5. On a vertical arrangement, at a certain distance on the right of the *S(1)* box, we place all the boxes ending the sequences which start with S(1), and we draw arrowhead lines from *S(1)* to each of them; the new boxes and lines are labelled.
6. The process is repeated for each of the newly represented states, but any state already in the diagram is not drawn again.

7. This is repeated until the list is finished.
8. This procedure can be elaborated and extended to include states in which there are messages hold in the buffers, to have state diagrams for objects with buffering capability.

The above sketched algorithm is only intended as the core of a procedure; it needs to be refined to produce graphic quality diagrams.

5. Deriving object diagrams from process diagrams

The contents of the preceding sections represents an analytical work useful to thoroughly understand the relations and mappings between equivalent processes and objects. Nevertheless, when we come to implementation, it may be more reasonable to take advantage of the fact that there already are tools that generate code from a given structure diagram, and all we need is to build a pre-processor to those tools that transforms a given process diagram into the service diagrams of its equivalent object.

To do this, we keep the process diagram unbroken and we try to derive from it the diagram of each of the object services, by selecting what is relevant to that service and suppressing what is not. That is, when we are trying to create the diagram for a service A which deals with message A, we have to suppress, from the process diagram, all the elements which are not compatible with the fact that a message A, has been read and is still available (not used up by the process). The elements to be suppressed are all the read operations that do not read A messages and all the parts of selections or iterations that are executed only when no message of type A has just been read or is still available for processing.

It is not obvious that we can determine when a record of a certain type, say A, is still available for processing just by inspecting a process diagram. There is no problem in seeing when it is read but, how do we find if it has been processed and is no longer available? We need to rely on some diagram labelling convention: there may be a rule that after a sub-tree labelled process A is exited, A is no longer available, or that this fact is indicated in some other way in the diagram.

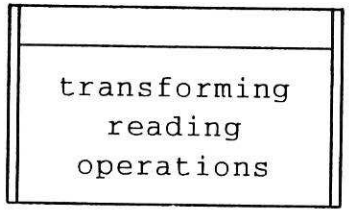
Once we have the process diagram customised with these suppressions, its transformation in the service diagram is immediate if we remember that every read operation (where a state is defined) is at the same time the end of one (or several) execution thread(s) and the beginning of other (or others), and if we have in mind the more general structure of services shown in figures 3.1.5.1 and 3.1.5.2. All we have to do is to "cut" the process diagram at all the points where a read operation is, and "hang" all those read operations from the outer selection which is the top structure in the service diagram; all these parallel branches of the service diagram will have an end of process at each of the next first read operations found in them, that is: the read operations will have not a follow element in the service diagram, and they will be translated into set state plus return operations.

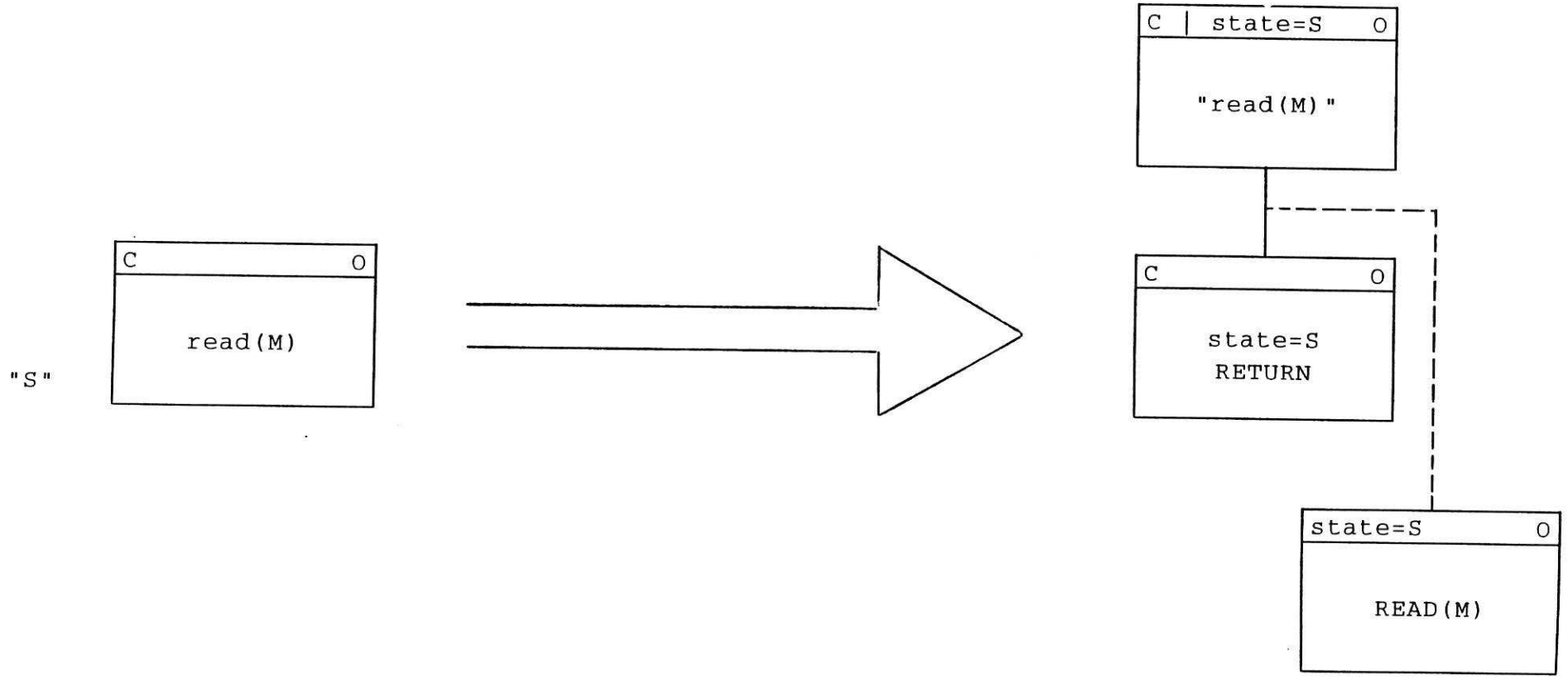
To avoid duplications of code, when we find execution sequences with a common trail (that is, when the same element, and the sequence it starts, belong to more than one of the service's outer selection branches), we have to combine them in the way shown in figures 3.1.5.4 and 3.1.5.5.

Before "cutting" the process diagram at the read operations, and because each of them has a double role as closing and starting element in the service execution sequences, we have to split each of the reading boxes in two, as figure 5.1 shows. The read operation is no longer an operation but a label, as the service is a sub-process called by other objects, and it does not execute any read operation on the incoming messages.

TRANSF.DAT

Figure 5.1.





35 - Figure 5.1 - transforming reading operations

To maintain consistency with the Jackson diagram notation, the condition C has been changed to C or (state=S); in fact, this box will not appear in the service diagram, and it does not mean that if this box is reached when state=S the dotted line is followed. The box with the state change and the return operations will be an end of process box in the diagram service, what means that the part of the process diagram which follows the return must not be taken into the service diagram. The dotted line from which the label READ(M) hangs does not indicate any real link, as execution of the branch starting at that state does not follow the previous state branch, but is initiated by the arrival of a new message; it only indicates where the element hanging from it is placed in the process diagram, so that its following element can be determined. In fact we have to "cut" the process diagram precisely at the dotted lines. Obviously, the box labelled READ(M) does not imply any read or any other kind of operation: it only signals the beginning of execution when the object is called with a message of this type and it is at state S.

Our first stage in obtaining service diagrams will be, then, to split all the read operations in the process diagram in the way explained in the preceding paragraph; this stage is common for all the services of a same object. Figures A.3.2.1 and A.3.2.2 show second process file ABC once this first transformation has been applied.

The second stage is specific to each service. It consists of suppressing, in the modified process diagram, all the elements not relevant to the processing of the message dealt with by this service. Figures A.3.3.1 and A.3.3.2 show second process file ABC after this second transformation has been performed in order to get the structure diagram of service A.

(The elimination, in figure A.3.3.2, of the box containing the operation state=XII again requires that it be possible to determine that the condition C available can not happen at that point while processing A. The logic to do it is that only two records are available to the process at any time; a B record has just been processed up, and an A must have read -- as we are within service A --, so no room is left for C. This type of logic, based on the records/messages read and still available, must be implemented in the tool that converts diagrams, as we have already pointed out.)

In these two figures, and to show how the structure of service A diagram is already implied in the process diagram, we have added arrows connecting each element to its following element in the service (except when it is very obvious because one element is a member of the first one's structure, as process current keys, which belongs to key cycle), and arrows coming from the service's top selection to each of the service's starting points that correspond to the different states; in this way, all the boxes in a same chain of arrows represent an execution sequence, which ends at the first RETURN found.

Now, we only have to draw the service diagram with the sequences indicated by the arrows. When a certain element E is a following element for more than one preceding element -- such as O, which is a following element for N and also for READ A when state=III -- all the sequences preceding it are members of a selection whose follow element is E -- O in this case --; this can be expressed correctly taking into account the equivalence between figures 3.1.5.4 and 3.1.5.5.

Figure A.3.5 shows the final diagram for service A; the transformation involving figures 3.1.5.4 and 3.1.5.5 has been applied recursively to avoid duplications of code.

In a similar way we obtain the diagrams of service INITIALISE, figure A.3.4, service B, figure A.3.6, service C, figure A.3.7, and service eof, figure A.3.8.

The iteration at key cycle is not present in the service diagrams, as it execution is always interrupted before processing can reach its end, and either re-enter the iteration or take its follow part.

Finally, duplications imposed by the structure of processes, can be avoided, in a way parallel to the one followed to get the object's code:

- first, when two, or more, logically identical pieces, such as take keys and take new keys are found in the process diagram -- once it has suffered the two required transformations and it is ready to be "cut" -- and they are placed in the same equivalent points -- immediately preceding process all A's and B's, in this case -- only one of them is needed; the others can be eliminated and all of its links connected to the one remaining
- second, if the several pieces are identical, but they are not placed at equivalent points -- as it would be the case for M and N, if they happened to be identical -- they can be made a subroutine when, at a later stage, the object diagrams are converted to code.

6. Deriving process diagrams from object diagrams

6.1. Conversion procedure

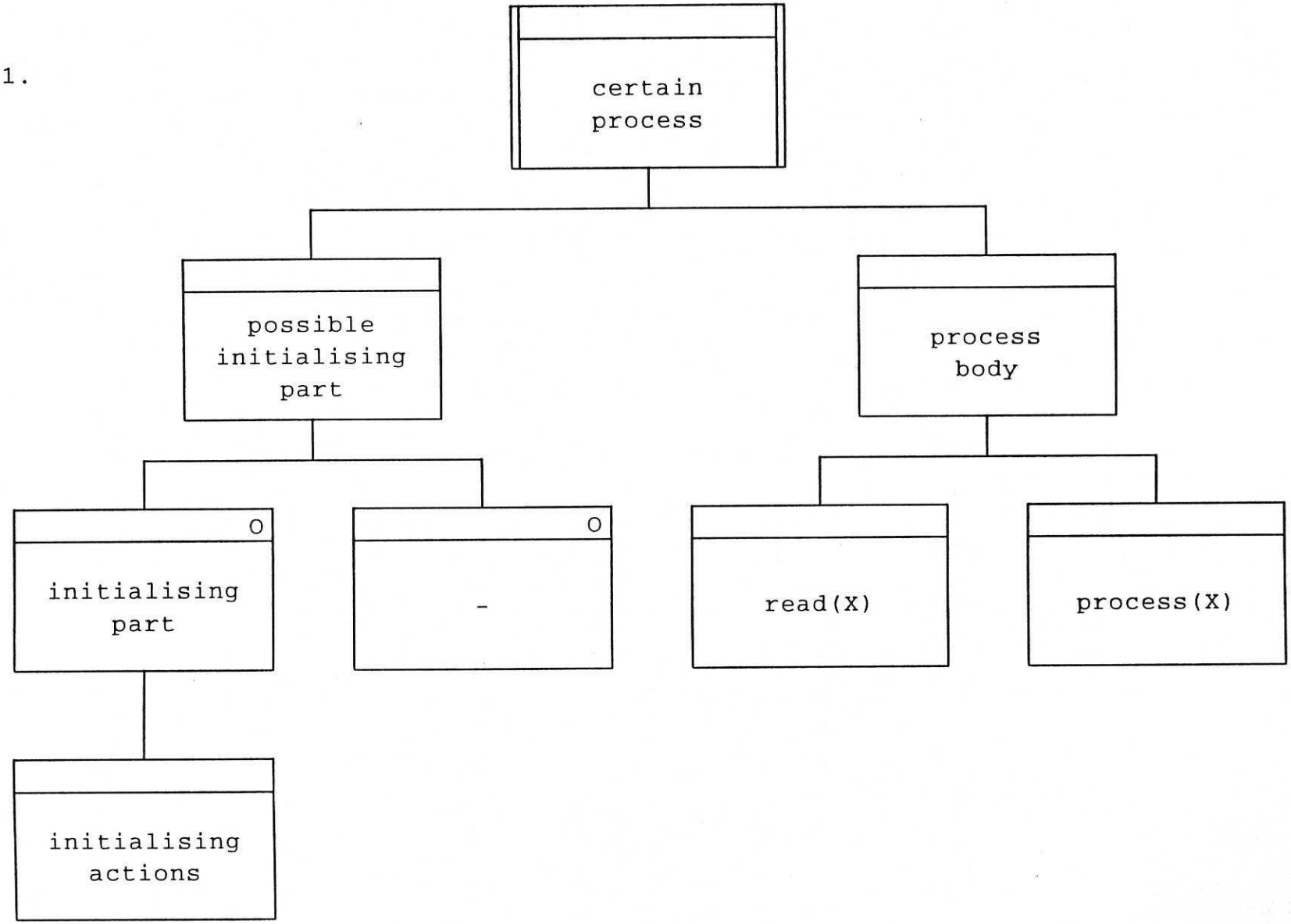
For the same reason that it is possible to extract the structure diagrams of an object's services from the diagram of its equivalent JSD process -- this is, because both entities, the process and the object, are made up of the same logical arrangements of operations, though interleaved in the process and separate in the object -- it is also possible to build up a process diagram if the service diagrams of its equivalent object are known.

We will show this reconstruction for the modified second process file ABC, starting our work from the services shown in figures A.3.4 to A.3.8.

A process may have an initialisation service or not, its general structure being as shown in figure 6.1.1. If there is such a service, we first take its diagram as the first executable part of the process diagram, as we have done in figure A.4.1 for the modified process file ABC. It must be noticed that there is an indeterminacy about where to place the element process A/B/C, which represents the remainder of the process diagram, as it could equally have been placed as shown in figure A.4.2, or at an intermediate level; we must come back to this point later.

1.DAT

Figure 6.1.1.



Then, the element process A/B/C when state=I is obviously a selection with the three possible components when A read do process A, when B read do process B and when C read do process C, as shown in figure A.4.3. The piece of diagram describing each of these branches is the branch corresponding to state=I in each of the diagrams service A, service B and service C respectively. Its addition to the process diagram will, in general lead to the inclusion of new reading elements corresponding to new states, in this case those corresponding to states II, III and IV.

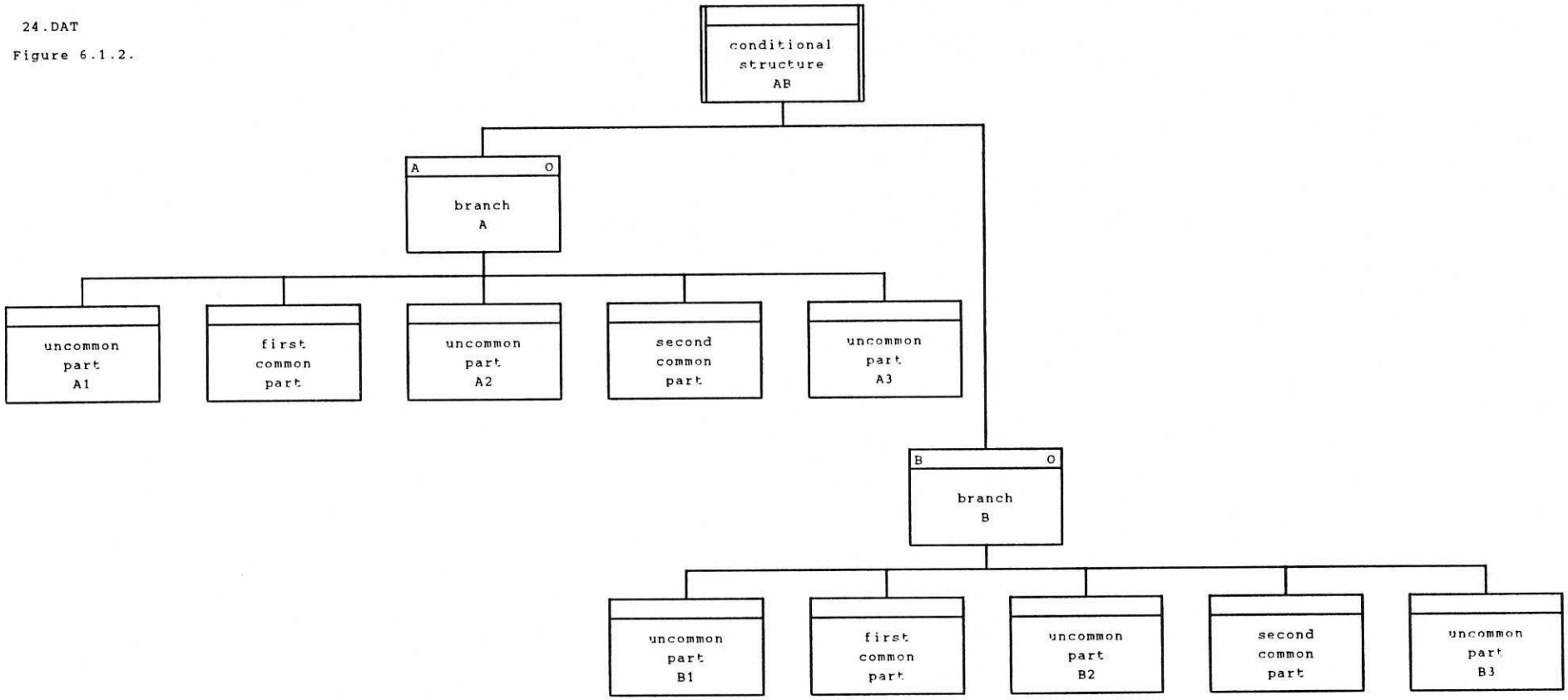
It may happen that all or some of those branches have a common initial or ending part, as is the case for the common initial elements M and read ahead second record in our example; the diagram can then be simplified by transforming the selection in a sequence made up of the common parts preceded, and/or followed, by a selection whose branches will have the uncommon elements only. This transformation is shown, for the general case, in figures 6.1.2 and 6.1.3; once the transformation is applied, figure A.4.3 becomes figure A.4.4.

The precedent procedure must now be applied to each of the processing boxes corresponding to the next states, II, III and IV. For instance, process A/B/eof at state=IV will be the selection shown in figure A.4.5; each of its branches is the branch for state=IV in the services for A, B and eof, respectively, and they are first separately shown in figures A.4.6, A.4.7 and A.4.8.

As for the conversion from processes to objects, the automatic procedure must keep track of which records/messages are read and available at any time, so that it "knows" when it is time to process when A read and C read at the bottom of figure A.4.5, for example, and, when it looks for the sub-diagram corresponding to state=IV in service A diagram, it can pick up only that part of this sub-diagram compatible with the current condition A read and C read. The reason why this selective pick up is needed is given in the next paragraph.

24.DAT

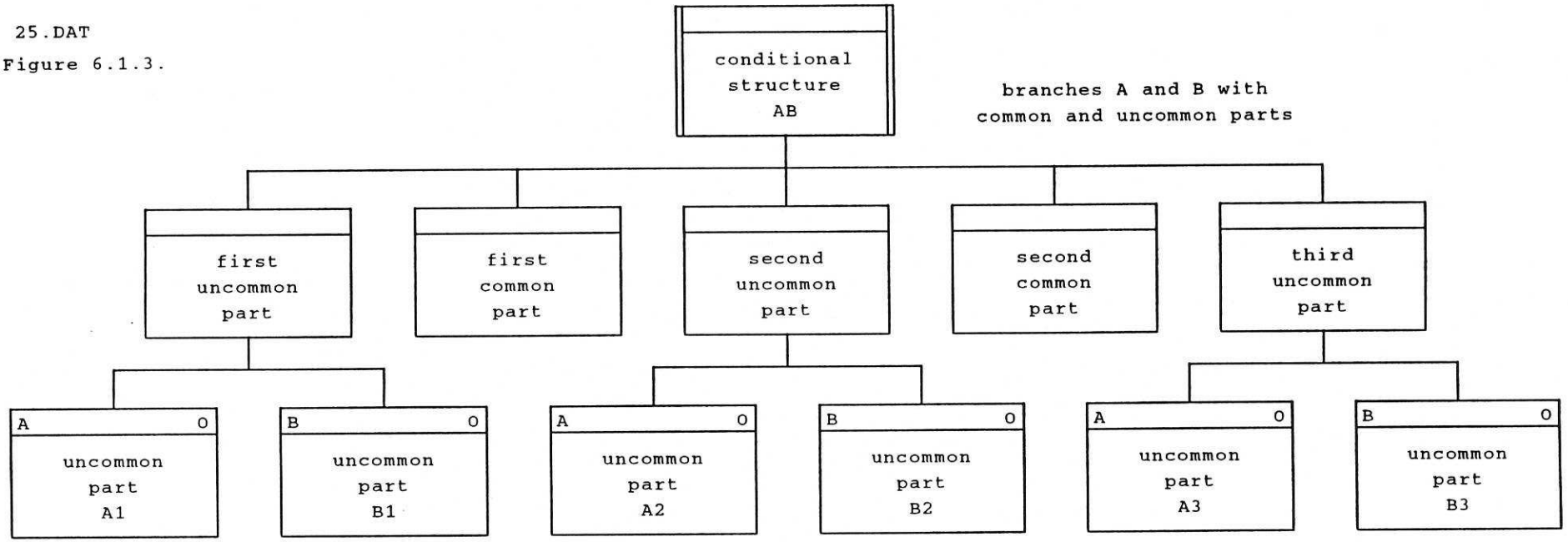
Figure 6.1.2.



40 - Figu 6.1.2 - conditional structure AB

25.DAT

Figure 6.1.3.



41 - Figu 6.1.3 - conditional structure AB

In a structure diagram, parts which are common to several execution sequences are drawn only once, but it is necessary to include appropriate control elements: this is the case for states III, IV, V, VI, IX and X in service A, figure A.3.4. However, only the parts of the service branch corresponding to state=IV, in service A, must be picked up as pertaining to process A at state=IV in the process, which implies to select only the parts of the diagram compatible with the condition A read and C read.

We can see in figure A.4.6 how only the parts where A and C are present have been taken from the sub-tree process current keys in service A. Similarly, in extracting from service B and service eof the process branches for B and eof at state=IV, only the parts corresponding to B read and C read and to C read and eof read, respectively, must be picked up, and we obtain the diagrams in figures A.4.7 and A.4.8.

Next we reduce the three branches of the selection in figure A.4.5 (which are shown in figures A.4.6, A.4.7 and A.4.8) to a simpler one, first realising that they have in common the initial sequence M, N and we obtain figure A.4.9. This can be further simplified by moving the selection down the process, what brings us to figure A.4.10. In this figure, each branch of the process current keys selection is subject to a double condition and it can be better represented by associating both conditions, as shown in figure A.4.11.

In this last diagram, figure A.4.11, there are conditions, such as [(ka<kb)&(AB|AC)], which are more complex than they were in the original process diagram of second process file ABC. In fact, in this process, ka<kb is equivalent to AB|AC, as it implies not(ka=max), this is: that an A message/record is available, for ka is set to max only when all As have been processed. But it might be the case that both conditions were incompatible, as it would happen if the left branch of process current keys in service A, service B and service C had the condition ka=kb, for example, which is not possible when A and C are present; the object diagrams would be inconsistent, and so would the derived process diagram be. As it is not possible to detect this without analysing the process operations, we will not attempt to simplify these combined conditions when they arise.

In a similar way we complete the other branches of figure A.4.4, process B/C at state=II, figure A.4.12, and process A/C at state=III, figure A.4.13, that we can incorporate into the process diagram. Then we obtain the diagram of figure A.4.14, in which repeated branches of the selections are not included..

Now we find a new situation. When we try to complete the diagram in figure A.4.14 for some of the new states, more precisely, for states V, VI, XI and XII, taking them from the appropriate service diagrams, we find that their trails comprise the compound element that contains process all A's and B's, key cycle, process current keys which, on the other hand, precedes all of them in the diagram in figure A.4.14. This means that there is an iteration in the process controlled by the condition while alfa and, after first incorporating into the diagram the processing for state=VII and simplifying, we obtain the diagram in figure A.4.15, in which there are no more new states for which processing needs to be defined, and is, then, the final process diagram.

6.2. Procedure evaluation

As we mentioned in relation to figures A.4.1 and A.4.2, there were several graphical points where we could attach the box for process A/B/C, but all of them were the same logical point: whichever we choose, process A/B/C will always be the following element to read A/B/C. This happens again with the diagram part following state=VII, and cannot be easily avoided unless additional conventions regarding diagrams are adopted, as the most adequate point to choose depends, otherwise, on the meaning of textual information. It is of no consequence to the logical value of the end diagram, but the labels can not be of any help in reading and understanding the diagram or any program derived from it; on the contrary, they can be confusing and misleading. Rules should then be adopted regarding diagram labelling that can help an automatic tool to properly place boxes when there are several equivalent possibilities, or labels should be ignored while converting objects to processes.

For the rest, the conditions controlling selections or iterations may be more complex than in the object diagrams; they must be, in any case, equivalent, unless the latter is wrongly designed

7. Implementation of inheritance in JSD

OOP concepts and techniques make it easy to modify classes by adding, deleting or redefining the class services. This is allowed precisely by the fact that services are coded as independent units, so that substituting one of them, for example, for a new one does in no way affect the others.

JSD process diagram are, properly speaking, class diagrams. Though Jackson and Cameron did not use the term class while describing JSD, a JSD process is defined as a type of process, many instances of which may be present in a same system. For instance, the process "customer" in a bank system will have as many instances as customers. The "customer" process structure diagram defines the type, class, "customer".

Nevertheless, JSD does not offer the flexibility of OOP to modify the classes and inheritance is a non existing concept in the method. Processes are each implemented as a single program, and it is not possible to change the code of a service independently of the rest of the program. Even if the change is done at structure diagram level, instead of at code level, the burden will still be considerable.

However, the transformations presented in previous sections offer a practical method to add, delete or redefine process services, and to implement inheritance, which allows for reuse in JSD.

For example, let us suppose that we have a structure diagram depicting a process P (a class) with services A, B, ...H. And let us try to derive a child class P' in which A is redefined as A', B is not present and a new service, X, is added. All we have to do is first to decompose the P structure diagram into its services, as we do to transform the process into an OOP object; then we remove A and B from the set of service diagrams, and we add A' and X to that set; finally we apply the reverse transformation, recomposing the P' diagram as when we convert an OOP object into a JSD process structure diagram.

(It seems obviously possible to transform the process diagram directly, without recourse to its decomposition and re-composition; such direct transformation, which we do not tackle here, should be based on the same principles used to convert processes into objects and vice versa.)

Still it will be necessary to generate code for the full inherited class P', while in the OOP implementation only A' and X code had to be produced. This may make the implementation of inheritance somewhat more cumbersome in JSD, but offers the opportunity to have reuse and keep the excellent specification characteristics of this method.

Proper implementation of inheritance in JSD by means of these mechanisms implies that we work with libraries of structure diagrams, and that all the programming is done at this graphical level, leaving to automated tools the task of code generation. This seems worth the cumbersomeness pointed out in the previous paragraph.

8. Conclusions

It seems possible to implement a JSD process, defined by its structure diagram, as an OOP object, using an OOP language, in an automatic way. This implementation can be performed by a computerised tool that derives the object's code from the process diagram, and we have described how such a tool could work. The automatic procedure is valid for processes reading a single (possibly rough-merged) data stream or several fixed-merged or data-merged data streams. In the latter case the object will have to be able to buffer input messages.

This tool would make possible to specify a system in JSD and implement it as a hierarchy of OOP objects, instead of one of inverted processes. This can be desirable for a number of reasons, for instance if a high degree of parallelism is needed and the object implementation allows for it (section 2.2bis); in this case, techniques to manage the allocation of services to different tasks or processors remain to be developed. The tool generating OOP code could also draw the state transition diagram of the process/object.

We have also shown how to convert process diagrams into object diagrams, and vice versa; this avoids the conversion to code which can be carried out later by using already existing tools.

When converting object diagrams into process diagrams, the object diagram must show, through some notation convention, when an input message processing ends. We have also seen that there is some indeterminacy in the aspect of the process diagram to draw, as a given box can some times be placed in several physical points which are logically the same. In fact, the aspect of the second process file ABC that we have derived from the object service diagrams looks somewhat different from the original we had used before, though its logic is the same. This may make the diagram labels not useful, even confusing, and deserves further consideration. It could possibly be avoided by enforcing additional notation rules.

The conversion of object diagrams into process diagrams would allow the implementation of systems specified in terms of OOD by means of the JSD implementation techniques, though we have not studied the conditions a specified system must meet to make it possible.

Finally, we have seen how application of both diagram transformations -- from processes to objects and vice versa -- permits to change the services of JSD processes (classes) by adding, deleting or redefining some of them, thus permitting the implementation of inheritance in JSD. This will permit the reuse of software while working with the excellent specification techniques of JSD and handling exclusively libraries that are made up classes graphically defined by the processes/objects structure diagrams.

9. Appendix. Examples

The following examples have been derived from the collating problem, such as dealt with by M. A. Jackson in "Principles of Program Design", section 4.2 "Collating".

Example 1. process files A and B

Figure A.1.1 shows the Jackson structure diagram of a program intended to collate two files, file A and file B, and figure A.1.2 shows the same structure with more detail. file A structure is depicted in figure A.1.3, and the structure of file B is similar.

file A is made up of records of type A, and file B of records of type B, and both are sorted in ascending order by a same key, k. If we call ka and kb the values of k in each file, and if max is the lowest number higher than all the possible keys (max>k), then the highest values of k present in each file will be kamax<max and kbmax<max. Not all the possible values of k need to be present in either A or B. The end of the file whose kmax is the lowest will be indicated by a record of a third type C; the end of the other file will be signalled by and eof record. If kamax and kbmax are equal, or if both files are empty, C is only present in file A.

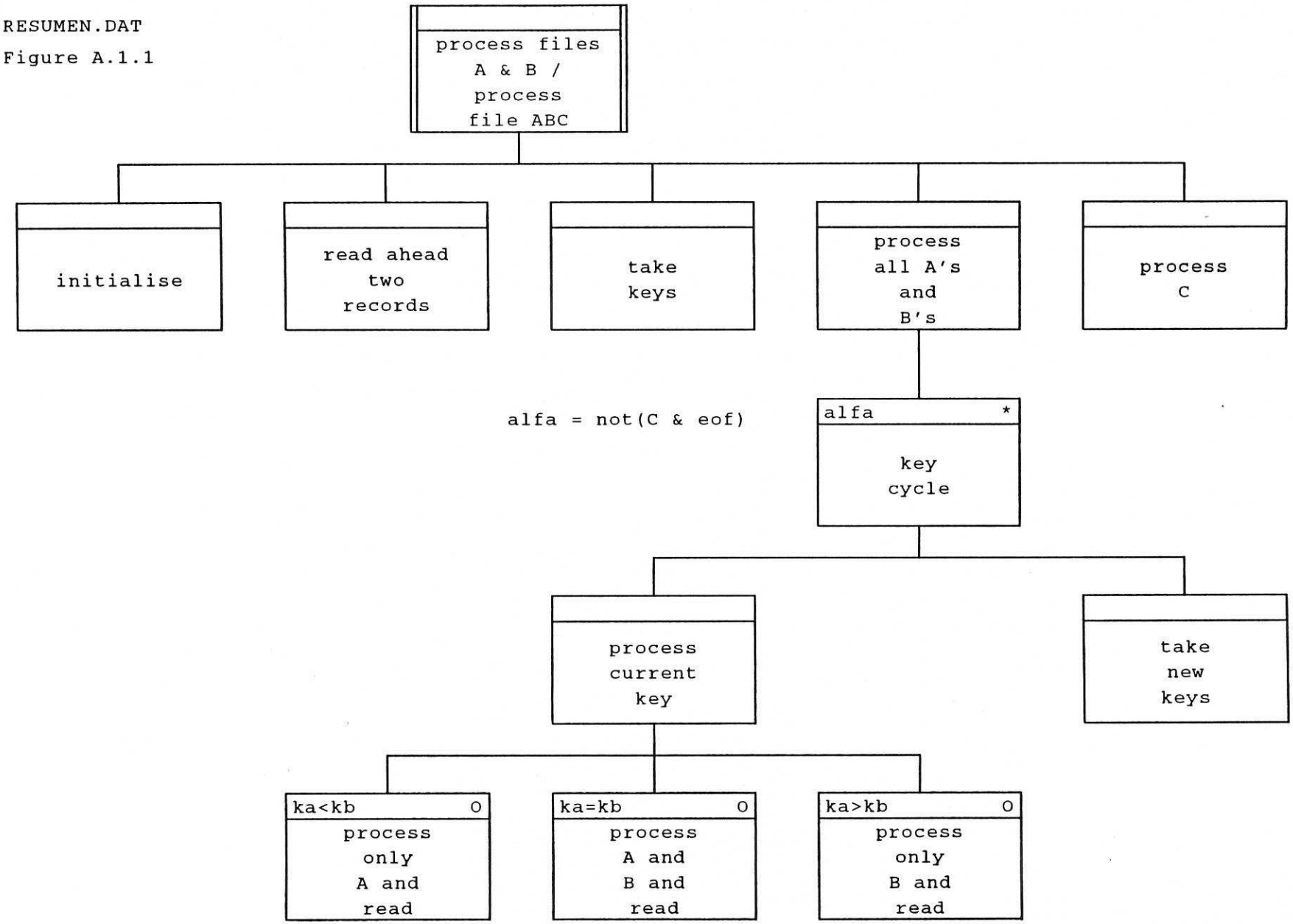
process files A and B reads one record from A and another from B, and processes whichever has the lowest value of k, or both if ka=kb. Then, it reads again from A or B, or from both, the same type of record, or records, that it has just processed. When it finds C, it detects that one type of record, A or B, is exhausted, and it processes all the remaining records of the other type.

It is worth to notice that, after having read A and B, and processed that with the lowest key, it will continue reading and processing records of that same type until it finds one whose key equals or exceeds the key in the non-processed record.

Figure A.1.1 also shows the structure of process file ABC, described in the section "Example 2" above.

RESUMEN.DAT

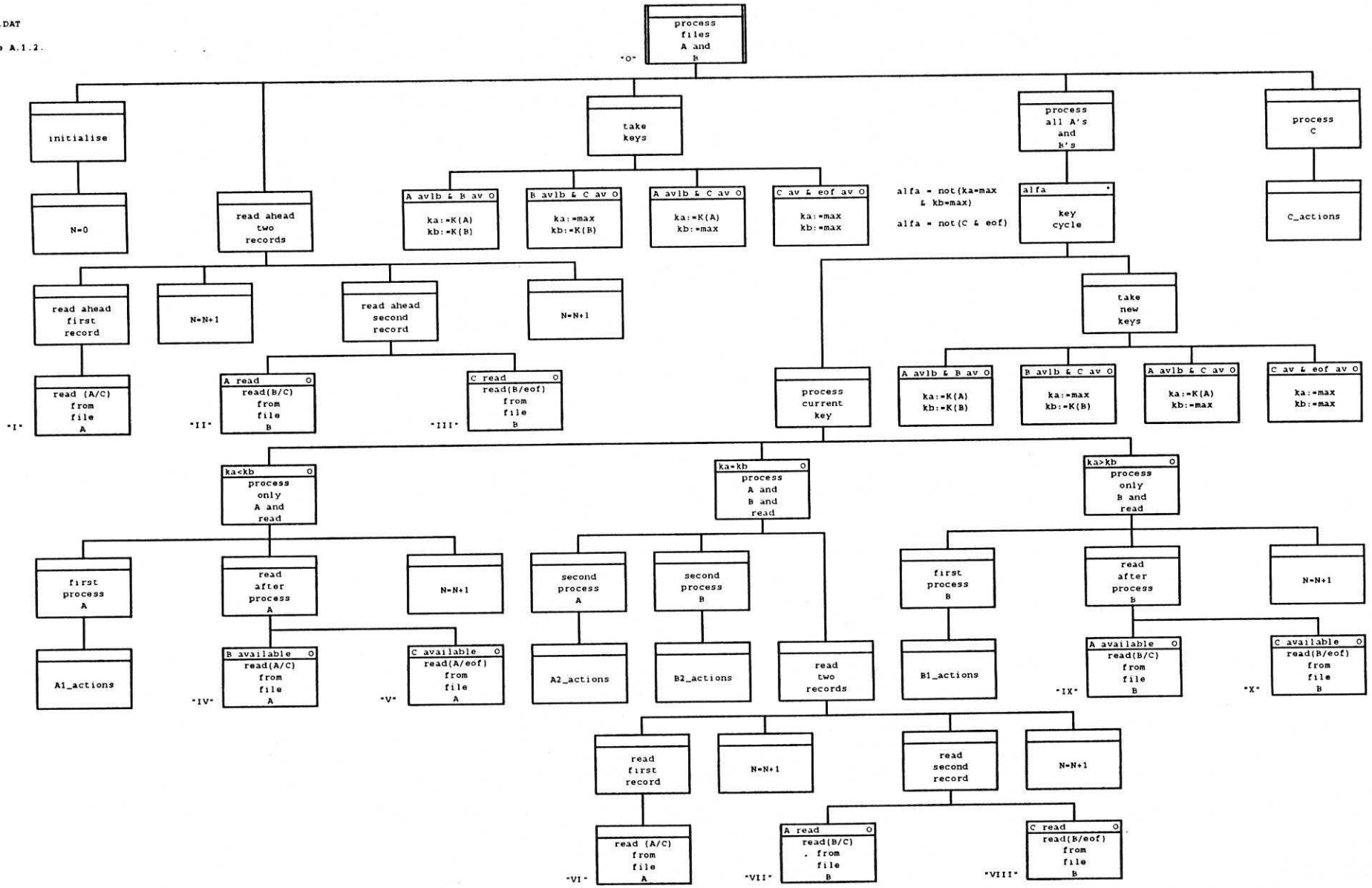
Figure A.1.1



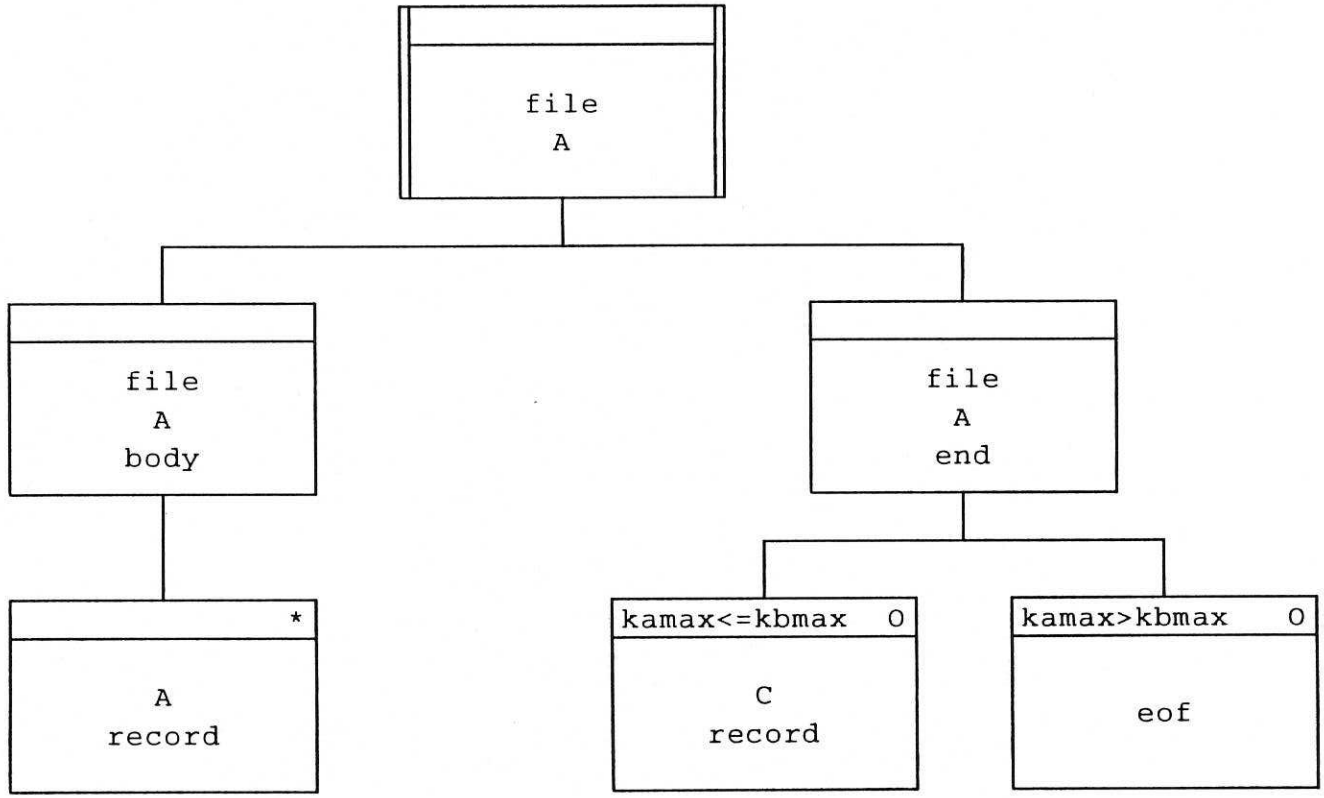
47 - Figu A.1.1 - process files A & B / process file ABC

A_&_B.DAT

Figure A.1.2.



FILEA.DAT
Figure A.1.3.



A records are in
increasing sequence by ka

Example 2. process file ABC

process file ABC, whose detailed structure diagram is in figures A.2.1.1 and A.2.1.2, is a program which performs exactly the same function as process files A and B, but reading from a single file, file ABC, in which the files A and B have been combined. The high level view of its structure is also the summary diagram shown in figure A.1.1 for process files A and B.

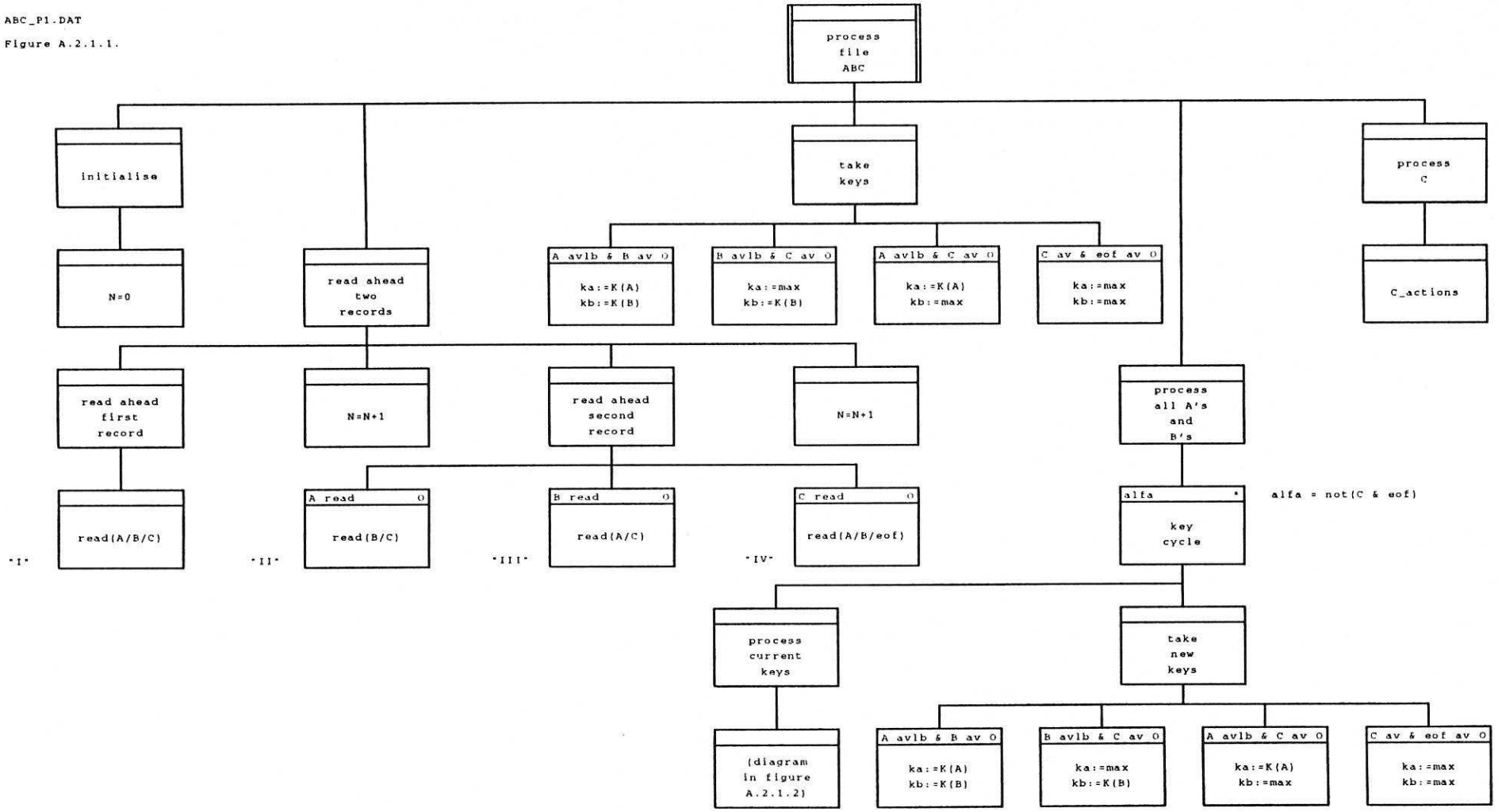
The structure of file ABC can be seen in figure A.2.2. It can be thought as if it had been written by process files A and B by sending each of its input records to file ABC as soon as it read them, without waiting to process them. In this way, A, B, C and eof records are in file ABC in the same order as they would have been read by process files A and B, and both of our programs will read the same records in the same order (but we could have designed process files A and B for it to start reading from file B, instead of from file A, and process file ABC caters for this possibility as well).

A reading operation like read A/B can never read a record different from A or B, as the program and input file structures are in perfect match. If, when implementing the program, we want to make sure that the input file has the assumed structure, then the read operations must check that the records read are of the expected type, and reject them otherwise.

Figure A.2.3 shows the text (code) of process file ABC in Jackson structured notation, and figure A.2.4 its State Transition Diagram. Finally, figure A.2.5 shows the text of a service belonging to an OOP object that implements the same function as this program. The discussions in the different sections of this study refer to all these figures when appropriate.

ABC_P1.DAT

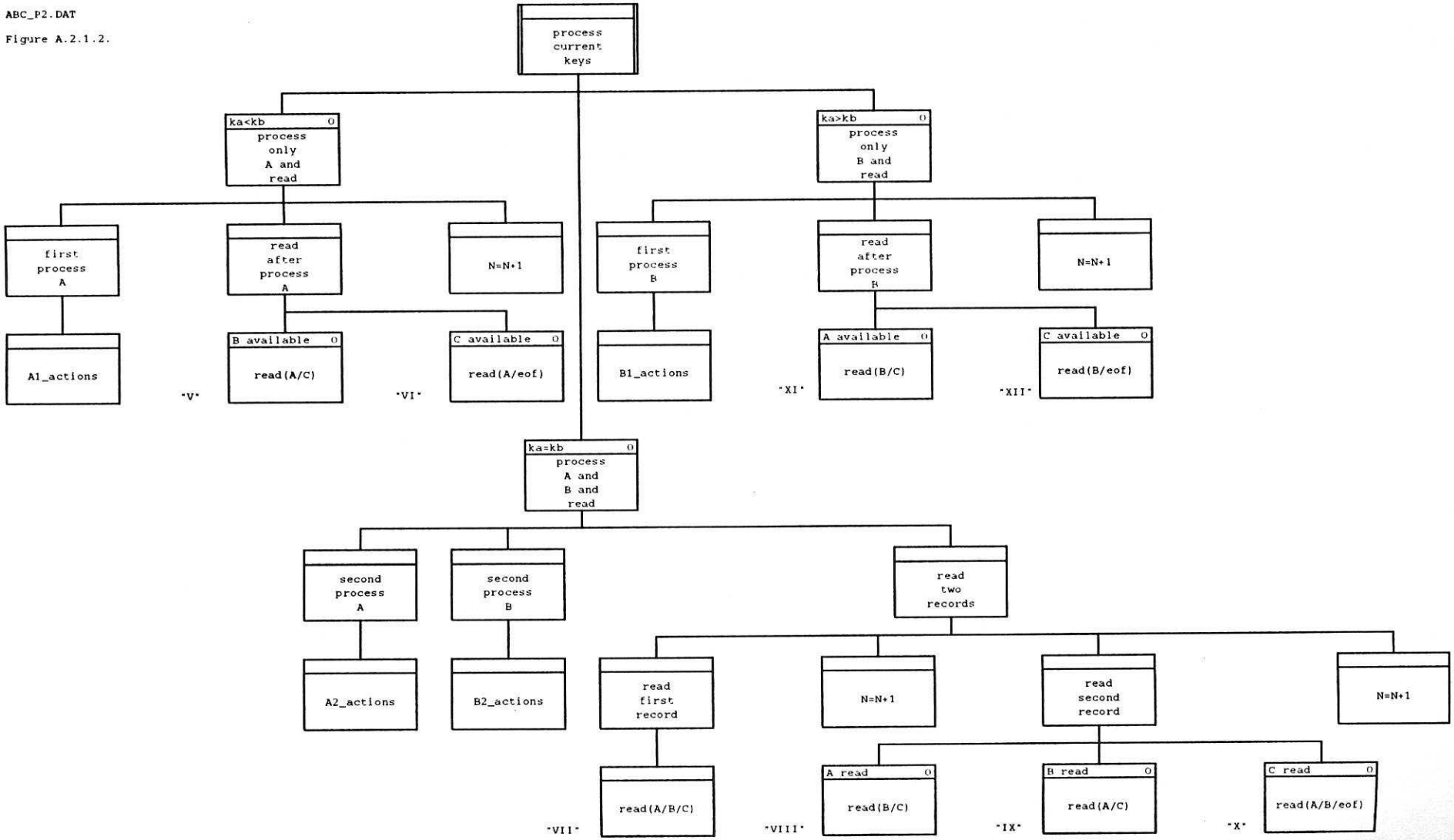
Figure A.2.1.1.



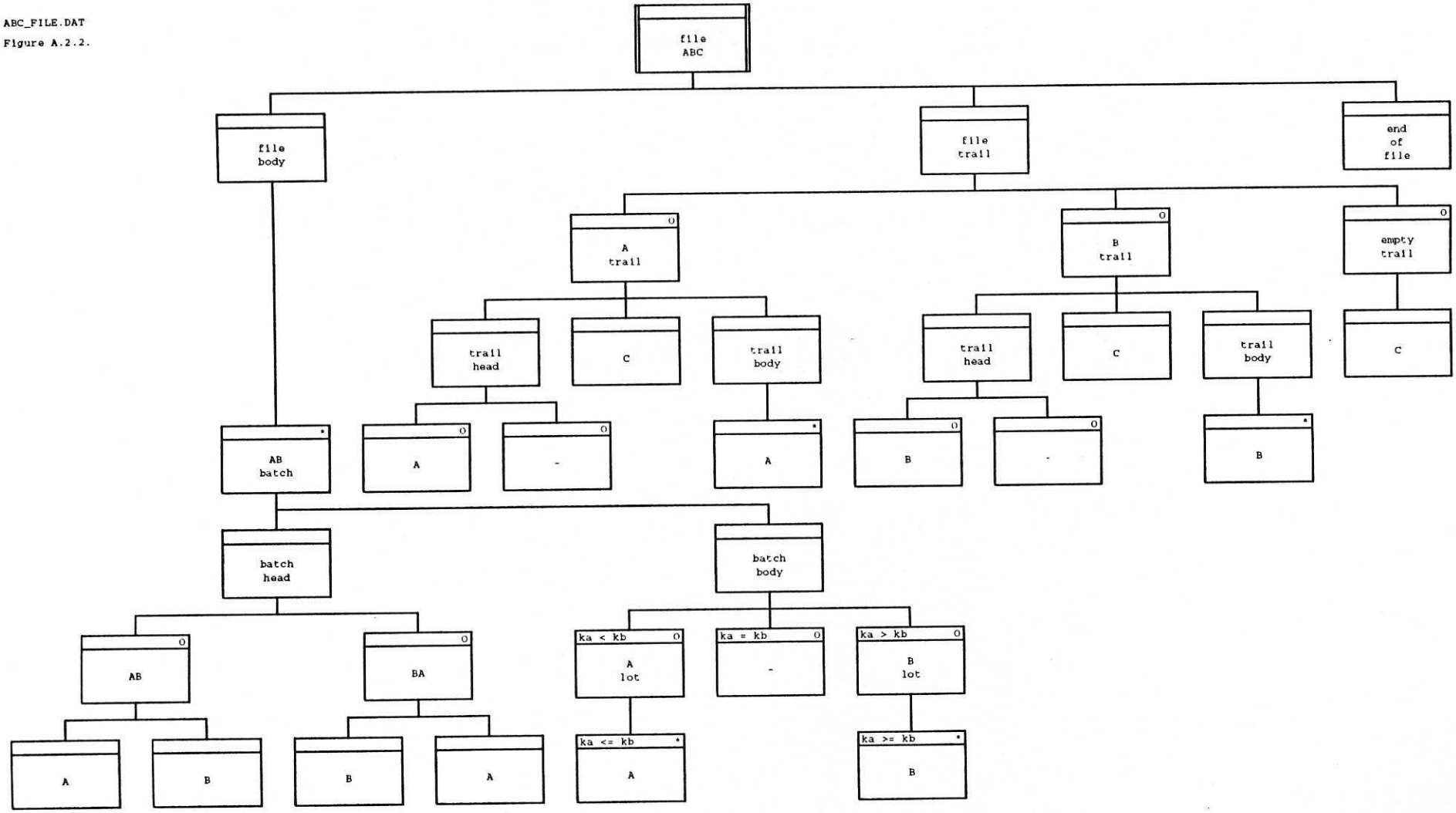
51 - Fi A.2.1.1 - process file ABC

ABC_P2.DAT

Figure A.2.1.2.



ABC_FILE.DAT
Figure A.2.2.



53 - Figu A.2.2 - file ABC

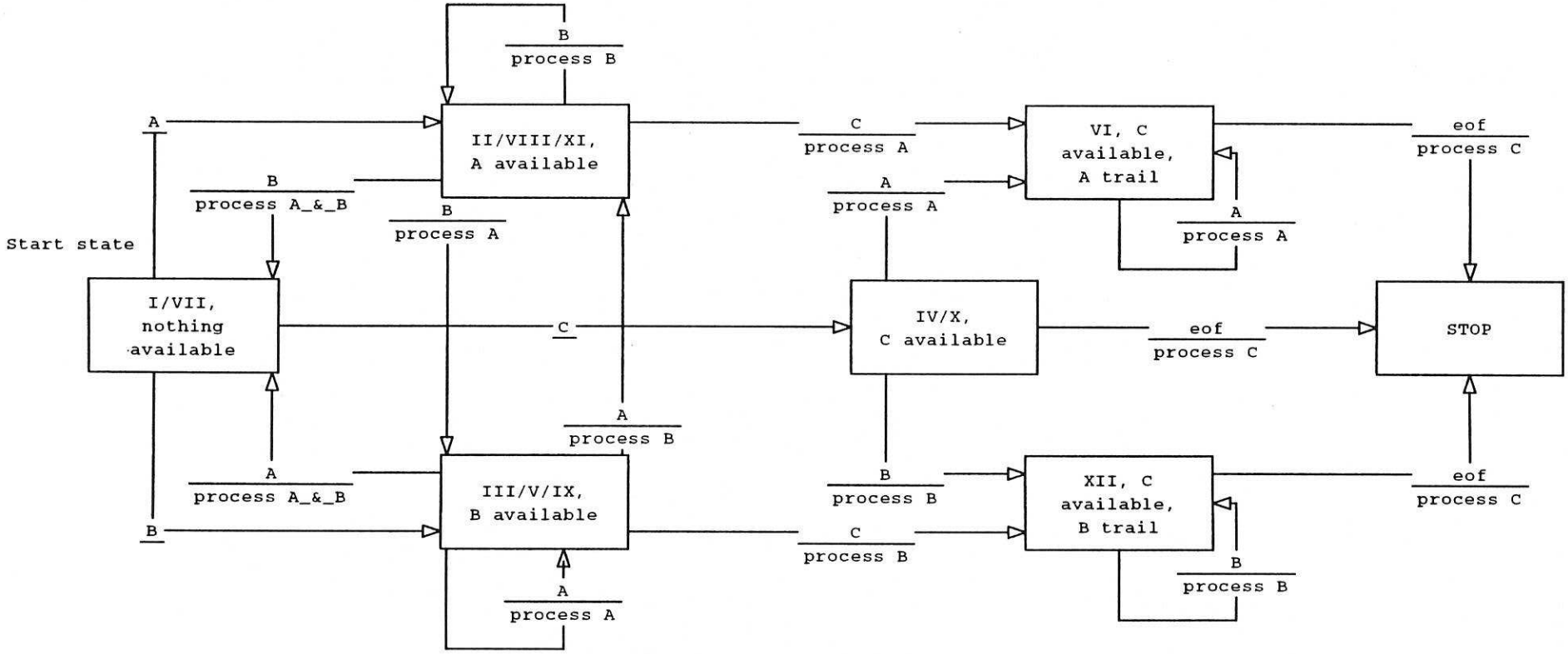
```
process_ABC_file seq
  initialise:                                    --state O--
  N:=0;
  read_ahead_two_records seq
    read_ahead_first_record:
    read(A/B/C):                                 --state I--
    N=N+1;
    read_ahead_second_record sel(A read)
      A_read:
      read(B/C);                                 --state II--
    read_ahead_second_record alt(B read)
      B_read:
      read(A/C);                                 --state III--
    read_ahead_second_record alt(C read)
      C_read:
      read(A/B/eof);                             --state IV--
    read_ahead_second_record end
    N:=N+1;
  read_ahead_two_records end
  take_keys sel(A available & B available)
    ka:=k(A);
    kb:=k(B);
  take_keys alt(B available & C available)
    ka:=max;
    kb:=k(B);
  take_keys alt(A available & C available)
    ka:=k(A);
    kb:=max;
  take_keys alt(C available & eof available)
    ka:=max;
    kb:=max;
  take_keys end
  process_all_A's_and_B's iter while(not(C & eof))
    not(C & eof):
    key_cycle seq
      process_current_key sel(ka<kb)                                  open process structure 1
        ka<kb:
        process_only_A_and_read seq
          first_process_A:
          A1_actions;
          read_after_process_A sel(B available)
            B_available:
            read(A/C):                           --state V--
          read_after_process_A alt(C available)
            C_available:
            read(A/eof);                         --state VI--
          read_after_process_A end
          N:=N+1;
        process_only_A_and_read end
      process_current_key alt(ka=kb)                                    process structure 1
        ka=kb:
        process_A_and_B_and_read seq                                    open process structure 2
          second_process_A:
          A2_actions;
          second_process_B:
          B2_actions;
          read_two_records seq                                          open process structure 3
            read_first_record:
            read(A/B/C);                         --state VII--              process line 1
            N:=N+1;                                          process line set 1
            read_second_record sel(A read)                   process line set 1   open process structure 4
              A_read:                                        process line set 1
              read(B/C);                         --state VIII-- process line set 1
            read_second_record alt(B read)                   process line set 1   process structure 4
              B_read:                                        process line set 1
              read(A/C);                         --state IX--  process line set 1
            read_second_record alt(C read)                   process line set 1   process structure 4
              C_read:                                        process line set 1
              read(A/B/eof);                     --state X--   process line set 1
            read_second_record end                           process line set 1   close process structure 4
            N:=N+1;                                          process line set 1
          read_two_records end                               process line set 1   close process structure 3
        process_A_and_B_and_read end                         process line set 1   close process structure 2
      process_current_key alt(ka>kb)                                                process structure 1
```

```
        ka>kb:
        process_only_B_and_read seq
          first_process_B:
          B1_actions;
          read_after_process_B sel(A available)
            A_available:
            read(B/C);                          --state XI--
          read_after_process_B alt(C available)
            C_available:
            read(B/eof);                        --state XII--
          read_after_process_B end
          N:=N+1;
        process_only_B_and_read end
      process_current_key end                                   close process structure 1
      take_new_keys sel(A available & B available)
        ka:=k(A);
        kb:=k(B);
      take_new_keys alt(B available & C available)
        ka:=max;
        kb:=k(B);
      take_new_keys alt(A available & C available)
        ka:=k(A);
        kb:=max;
      take_new_keys alt(C available & eof available)
        ka:=max;
        kb:=max;
      take_new_keys end
    key_cycle end
  process_all_A's_and_B's end
 C & eof:
  process_C:
  C_actions
process_ABC_file end
```

Figure A.2.3 -- process file ABC structured text

ABC_STD.DAT

Figure A.2.4.



```
message_A sel(status=I)
    N=N+1;
    read_ahead_second_record sel(A read):
      A_read:
      status:=II;
      return;                                  --state II next--
    read_ahead_second_record end:
message_A alt(status=III or status=IV)
    read_ahead_second_record end:
    N:=N+1;
  read_ahead_two_records end:
  take_keys sel(A available & B available)
    ka:=k(A);
    kb:=k(B);
  take_keys alt(A available & C available)
    ka:=k(A);
    kb:=max;
  take_keys end
   process_all_A's_and_B's iter while(not(C & eof))                    *1 begin*
    not(C & eof):
    key_cycle seq
      process_current_key sel(ka<kb)
        ka<kb:
        process_only_A_and_read seq
          first_process_A:
          A1_actions;
          read_after_process_A sel(B available)
            B_available:
            status:=V;
            return;                            --state V next--
          read_after_process_A alt(C available)
            C_available:
            status:=VI;
            return;                            --state VI next--
          read_after_process_A end
        process_only_A_and_read end
      process_current_key alt(ka=kb)
        ka=kb:
        process_A_and_B_and_read seq                                   open process structure 2
          second_process_A:
          A2_actions;
          second_process_B:
          B2_actions;
          read_two_records seq                                         open process structure 3
            read_first_record:
            status:=VII;                                                    service line 1
            return;                            --state VII next--           service line 2
         read_two_records end                                          close process structure 3
        process_A_and_B_and_read end                                   close process structure 2
      process_current_key alt(ka>kb)
        ka>kb:
        process_only_B_and_read seq
          first_process_B:
          B1_actions;
          read_after_process_B sel(A available):
            A_available:
            status:=XI;
            return;                            --state XI next--
           read_after_process_B end:
        process_only_B_and_read end
      process_current_key end
    key_cycle end
  process_all_A's_and_B's end
process_ABC_file end:                                                  *1 end*
message_A alt(status=V or status=VI)
          read_after_process_A:
          N:=N+1;
        process_only_A_and_read end:
      take_new_keys sel(A available & B available)                     *2 begin*
        ka:=k(A);
        kb:=k(B);
      take_new_keys alt(A available & C available)
        ka:=k(A);
        kb:=max;
```

```
    take_new_keys end                                                    *2 end*
  key_cycle end:
 process_all_A's_and_B's iter while(not(C & eof))                        *1 begin*
  not(C & eof):
  key_cycle seq
    process_current_key sel(ka<kb)
      ka<kb:
      process_only_A_and_read seq
        first_process_A:
        A1_actions;
        read_after_process_A sel(B available)
          B_available:
          status:=V;
          return;                          --state V next--
        read_after_process_A alt(C available)
          C_available:
          status:=VI;
          return;                          --state VI next--
        read_after_process_A end
      process_only_A_and_read end
    process_current_key alt(ka=kb)
      ka=kb:
      process_A_and_B_and_read seq
        second_process_A:
        A2_actions;
        second_process_B:
        B2_actions;
        read_two_records seq
          read_first_record:
          status:=VII;
          return;                          --state VII next--
        read_two_records end
      process_A_and_B_and_read end
    process_current_key alt(ka>kb)
      ka>kb:
      process_only_B_and_read seq
        first_process_B:
        B1_actions;
        read_after_process_B sel(A available):
          A_available:
          status:=XI;
          return;                          --state XI next--
        read_after_process_B end:
      process_only_B_and_read end
    process_current_key end
  key_cycle end
 process_all_A's_and_B's end
process_ABC_file end:                                                    *1 end*
message_A alt(status=VII)
          N:=N+1;
          read_second_record sel(A read):
            A_read:
            status:=VIII;
            return;                        --state VIII next--
          read_second_record end:
message_A alt(status=IX or status=X)
          read_second_record end:                                            service label 4
          N:=N+1;
        read_two_records end:                                                service label 3
      process_A_and_B_and_read end:                                          service label 2
    process_current_key end:                                                 service label 1
    take_new_keys sel(A available & B available)                         *2 begin*
      ka:=k(A);
      kb:=k(B);
    take_new_keys alt(A available & C available)
      ka:=k(A);
      kb:=max;
    take_new_keys end                                                    *2 end*
  key_cycle end:
 process_all_A's_and_B's iter while(not(C & eof))                        *1 begin*
  not(C & eof):
  key_cycle seq
    process_current_key sel(ka<kb)
      ka<kb:
```

```
        process_only_A_and_read seq
          first_process_A:
          A1_actions;
          read_after_process_A sel(B available)
            B_available:
            status:=V;
            return;                          --state V next--
          read_after_process_A alt(C available)
            C_available:
            status:=VI;
            return;                          --state VI next--
        read_after_process_A end
      process_only_A_and_read end
    process_current_key alt(ka=kb)
      ka=kb:
      process_A_and_B_and_read seq
        second_process_A:
        A2_actions;
        second_process_B:
        B2_actions;
        read_two_records seq
          read_first_record:
          status:=VII;
          return;                            --state VII next--
        read_two_records end
      process_A_and_B_and_read end
    process_current_key alt(ka>kb)
      ka>kb:
      process_only_B_and_read seq
        first_process_B:
        B1_actions;
        read_after_process_B sel(A available):
          A_available:
          status:=XI;
          return;                            --state XI next--
        read_after_process_B end
      process_only_B_and_read end
    process_current_key end
  key_cycle end
process_all_A's_and_B's end
process_ABC_file end:                                              *1 end*
message_A end
```

Figure A.2.5 -- message_A service structured text

Example 3. second process file ABC and its transformation into an object

second process file ABC, that can be seen in figures A.3.1.1 and A.3.1.2, is similar to process file ABC, but the operations N=N+1 -- which appear in many places in the latter -- have been changed into different operations, named M, O, P, Q, etc., and a new one, N, has been added.

Figure A.3.1.1 and A.3.1.2 are marked original diagram, for they are used to derive the service diagrams of the equivalent OOP object.

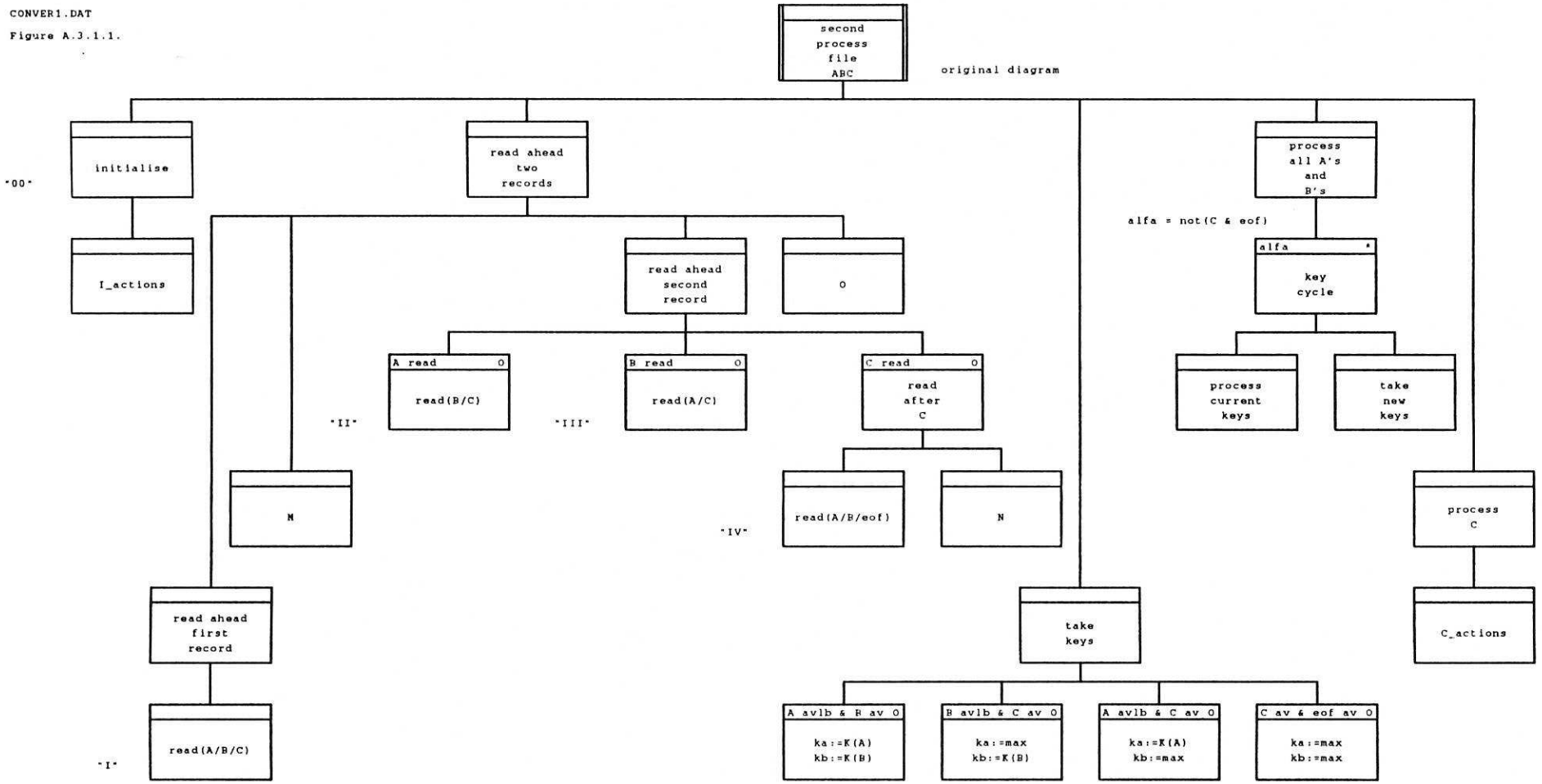
Figures A.3.2.1 and A.3.2.2, marked diagram after first transformation, show the result of applying the first transformation required to obtain the object service diagrams; this first transformation is the same on the way to derive any of the different services, so these two diagrams are further subjected to as many second transformations as service diagrams are to be obtained.

Figures A.3.3.1 and A.3.3.2, marked diagram after second transformation, show the precedent diagrams once the second transformation has been applied in order to obtain service A.

Finally, figures A.3.4 through A.3.8 show the structure diagrams of the object service INITIALISE, service A, service B, service C and service eof.

CONVER1.DAT

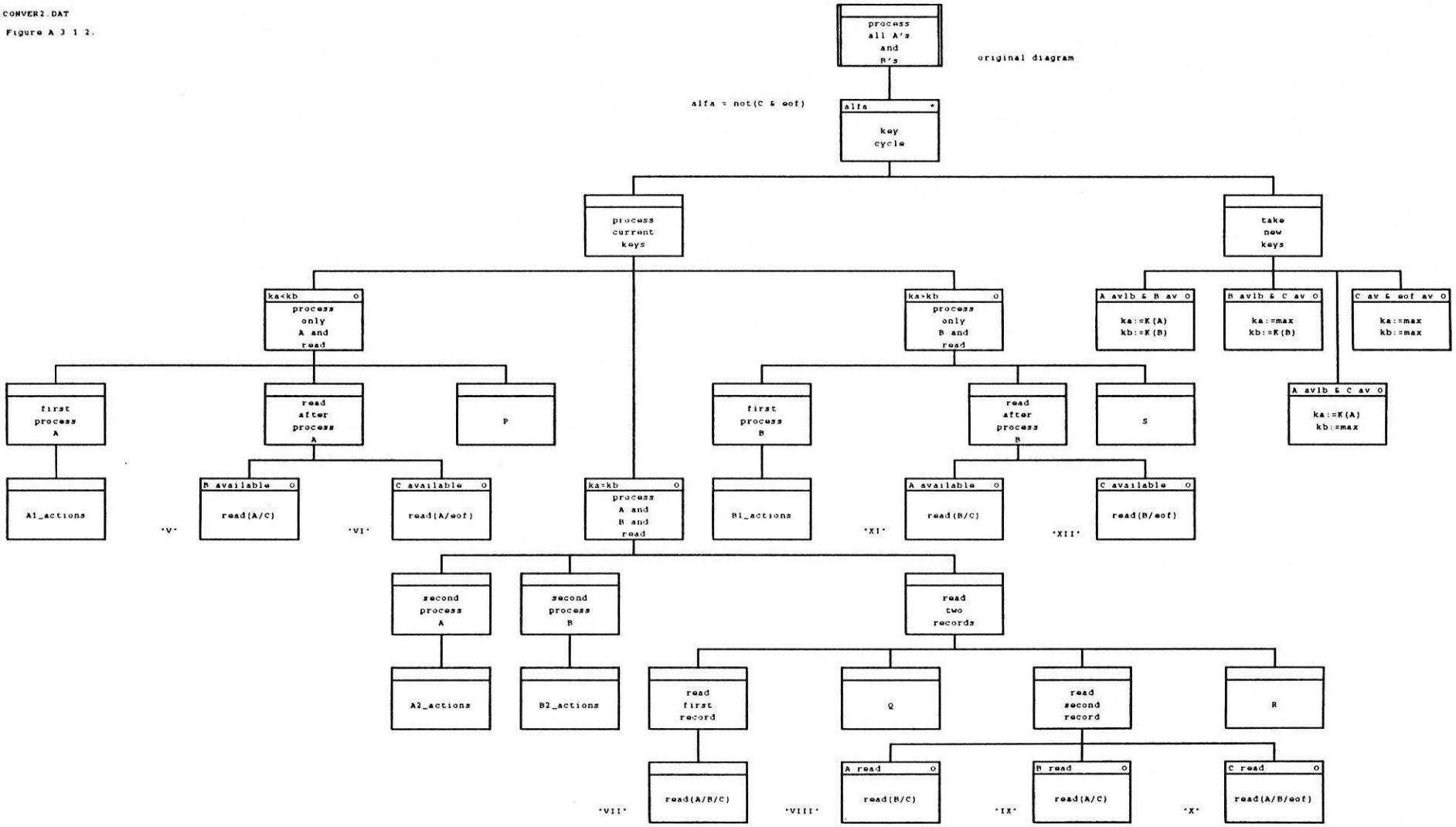
Figure A.3.1.1.



61 - Fi A.3.1.1 - second process file ABC

CONVER2.DAT

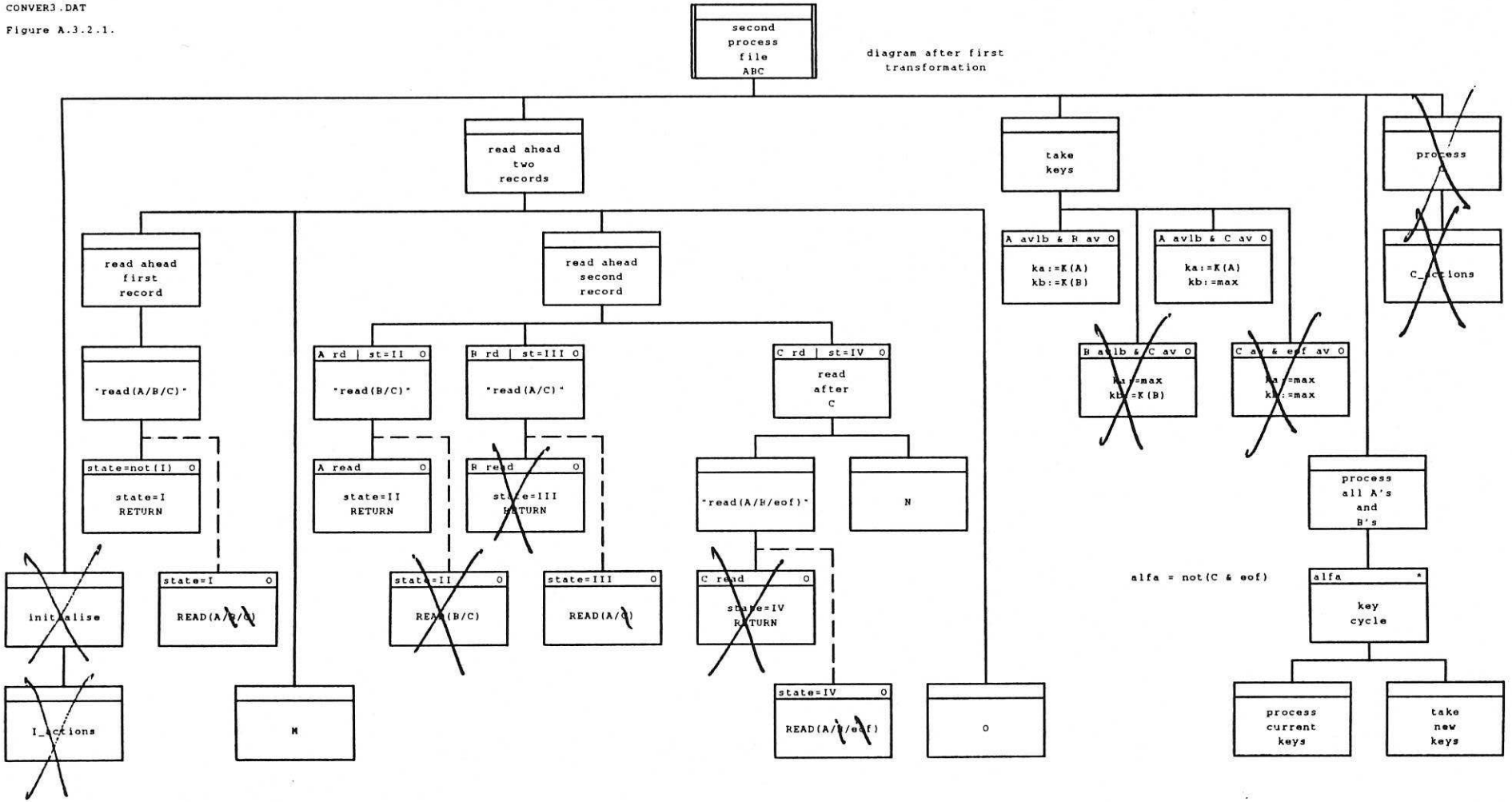
Figure A 3 1 2.



62 - Fi A.3.1.2 - process all A's and B's

CONVER3.DAT

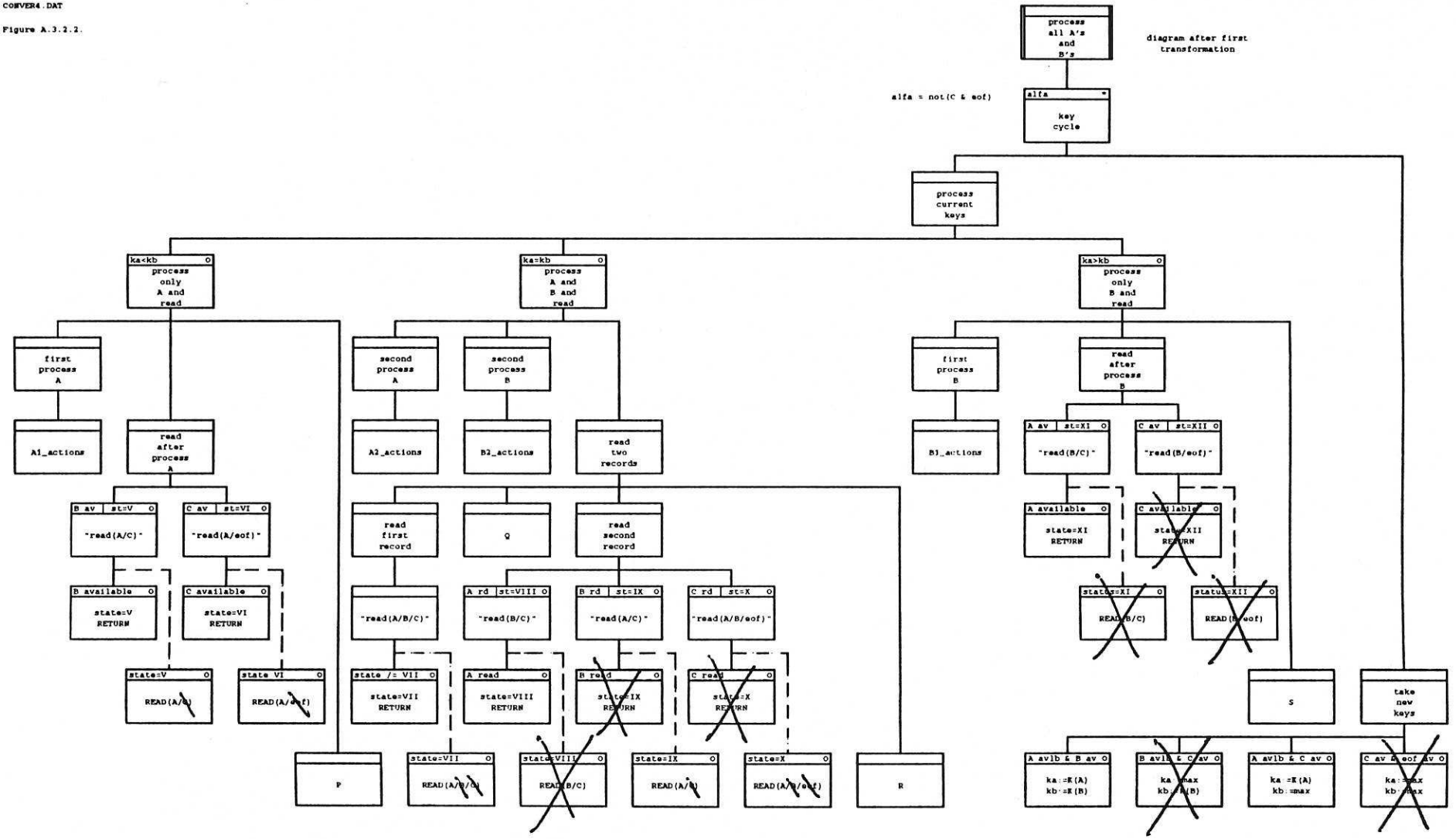
Figure A.3.2.1.



63 - Fi A.3.2.1 - second process file ABC

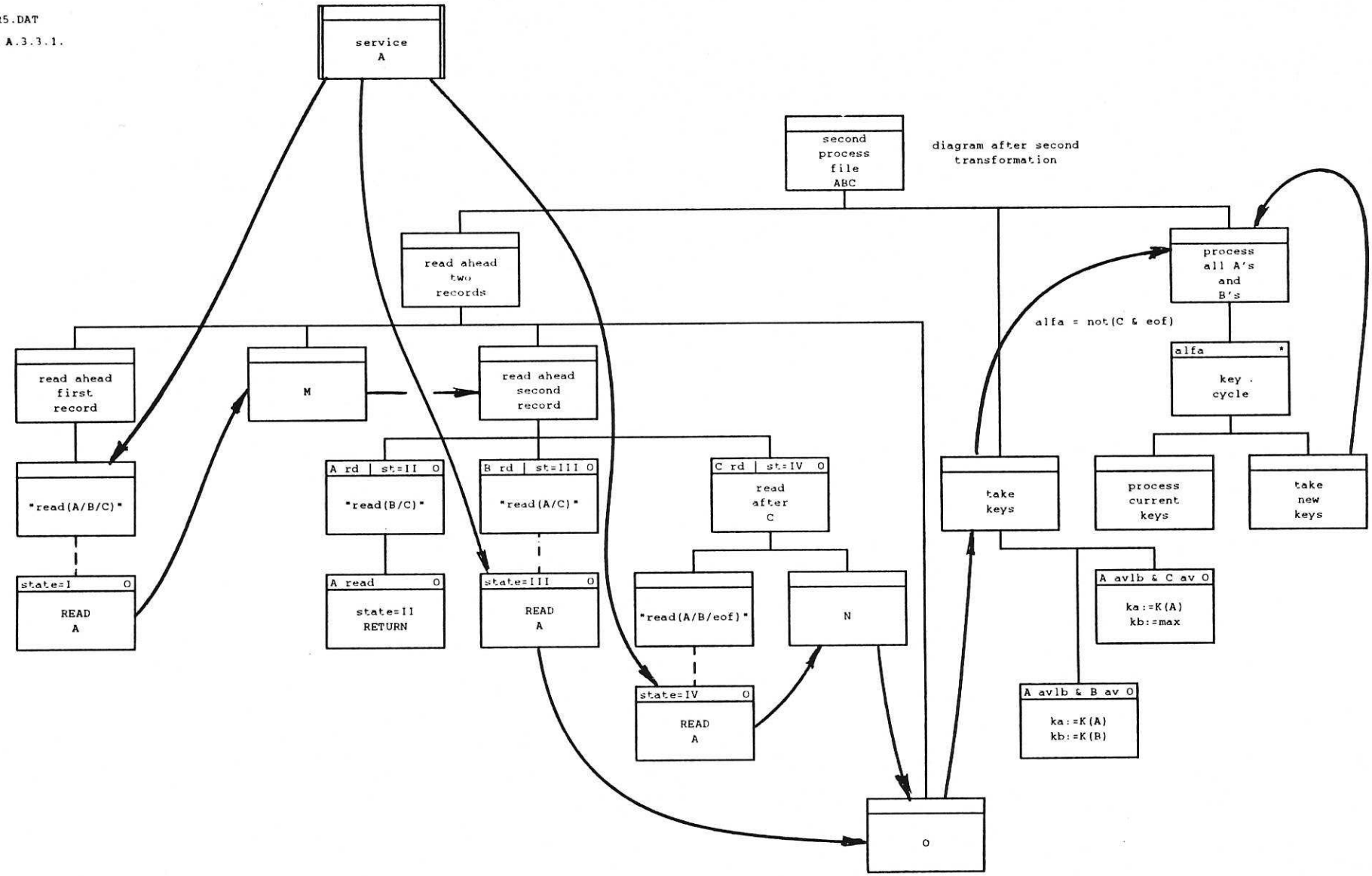
CONVER4.DAT

Figure A.3.2.2.

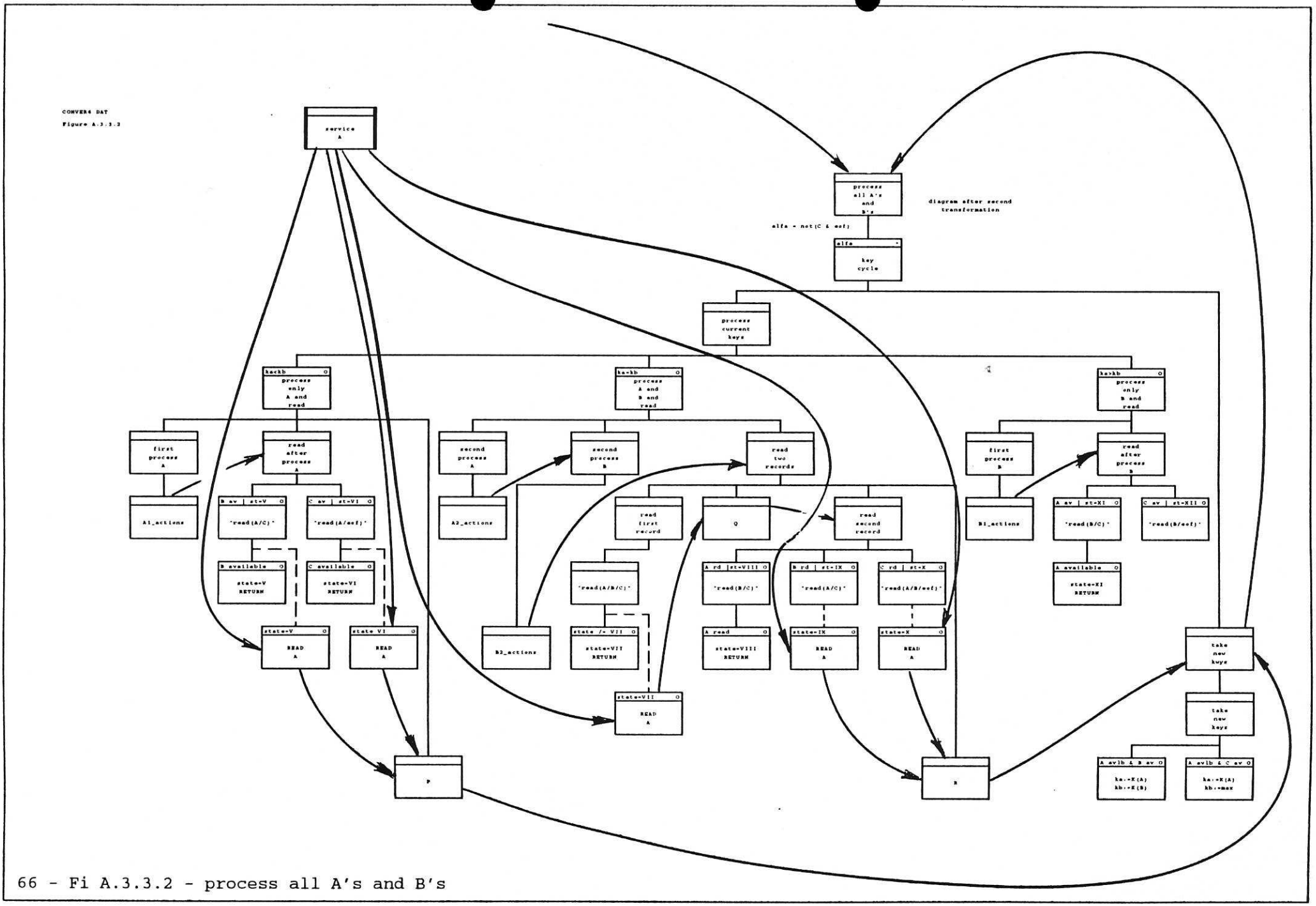


CONVER5.DAT

Figure A.3.3.1.



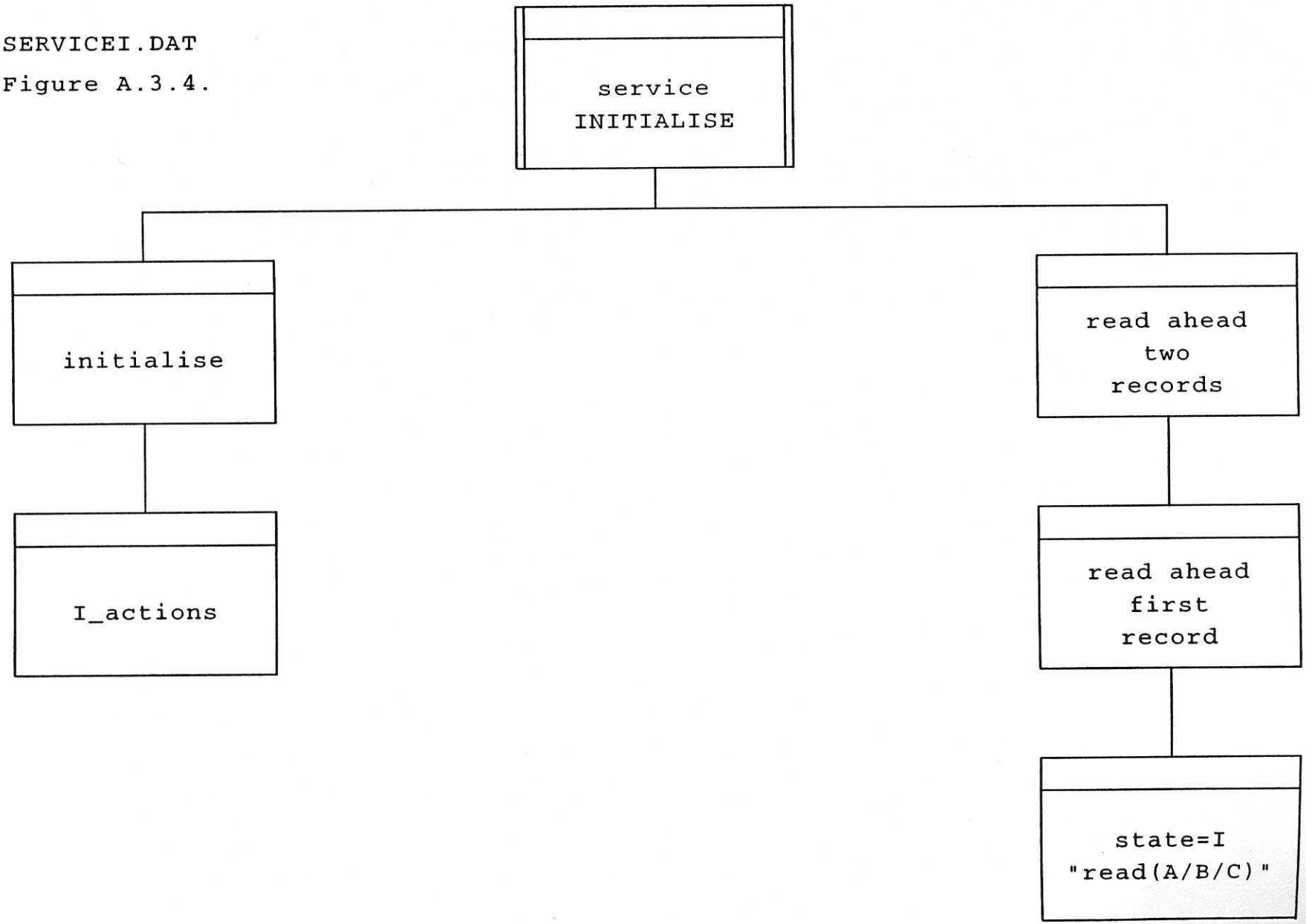
65 - Fi A.3.3.1 - second process file ABC



66 - Fi A.3.3.2 - process all A's and B's

SERVICEI.DAT

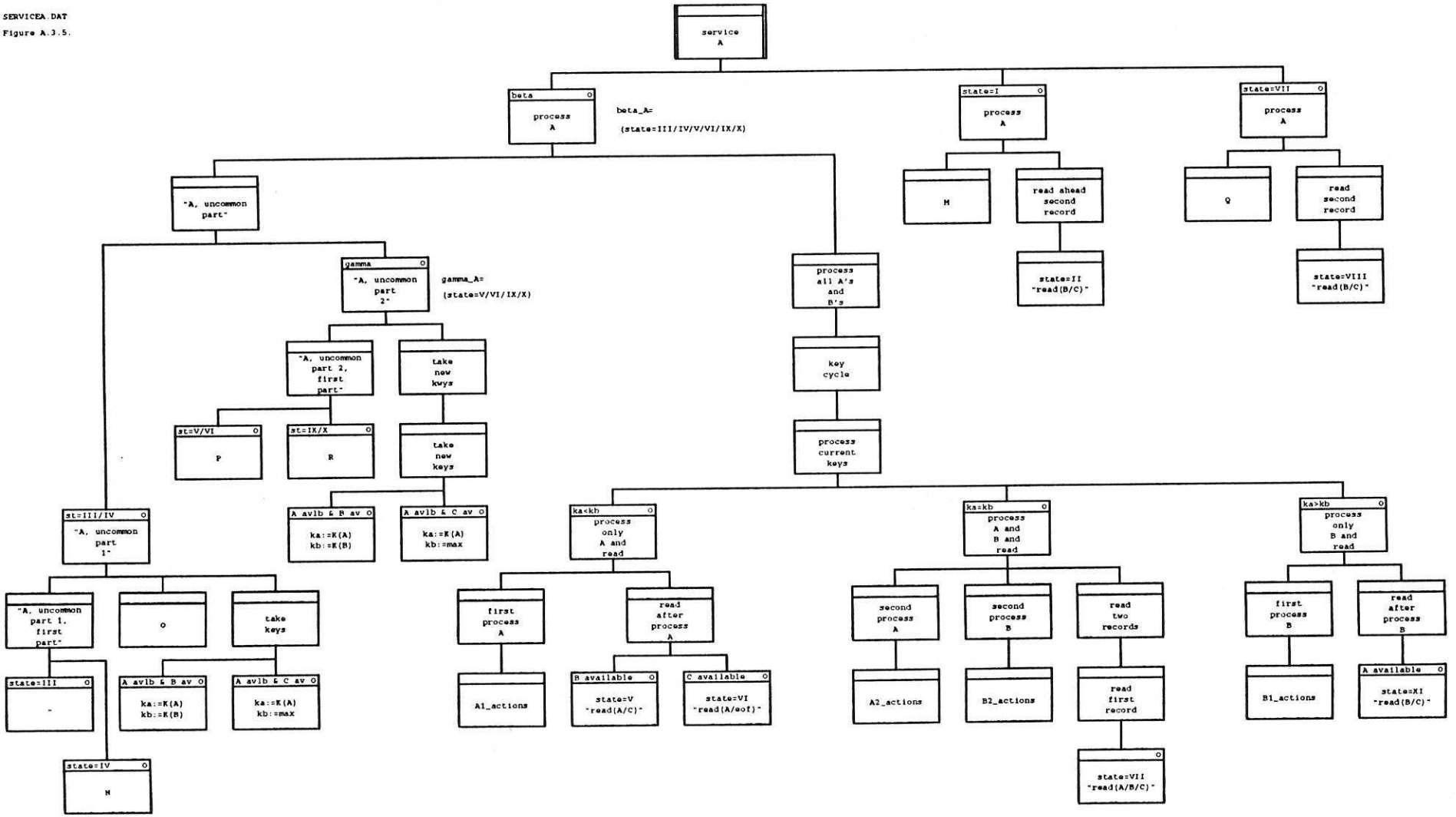
Figure A.3.4.



67 - Figu A.3.4 - service INITIALISE

SERVICEA.DAT

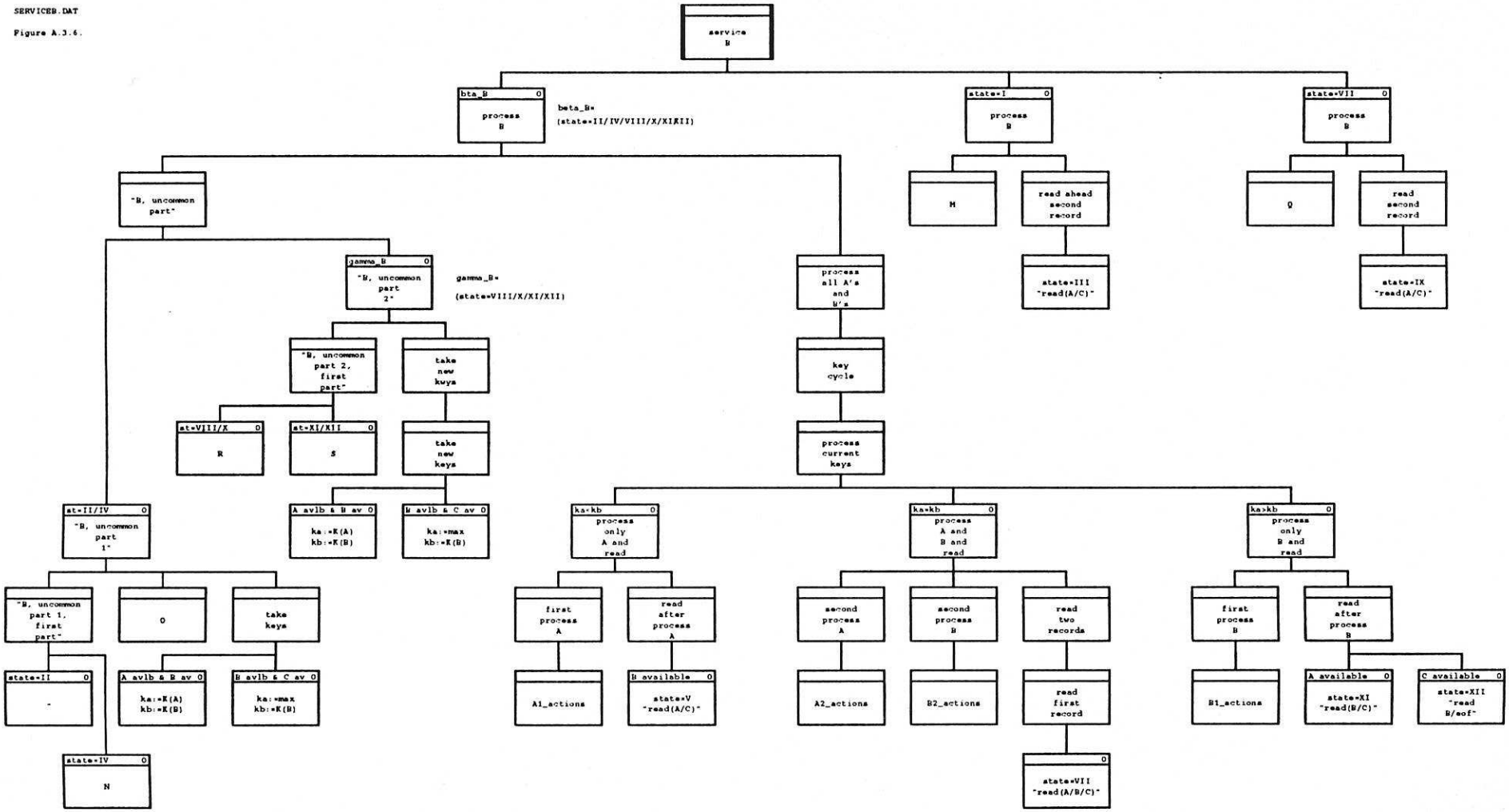
Figure A.3.5.



68 - Figu A.3.5 - service A

SERVICEB.DAT

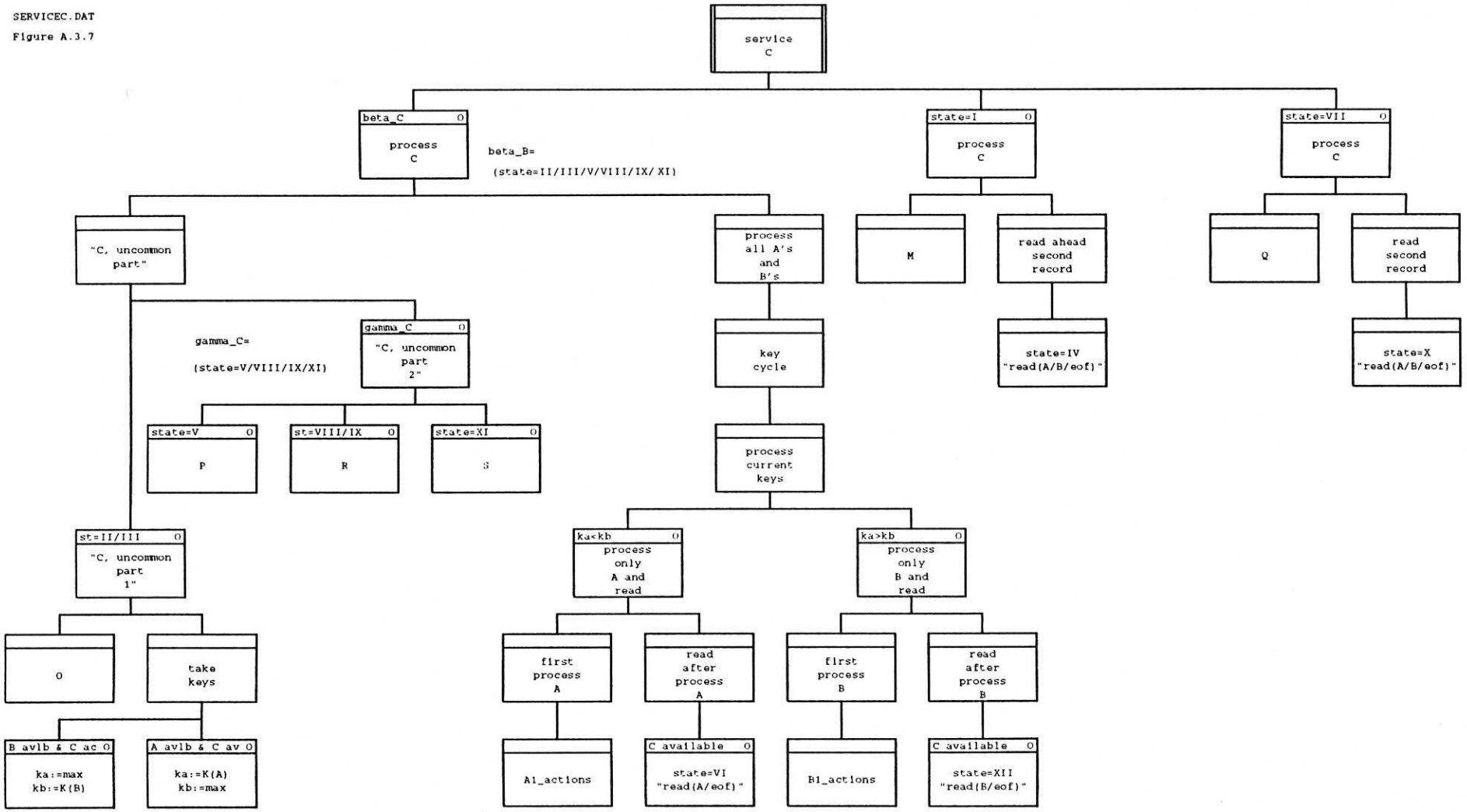
Figure A.3.6.



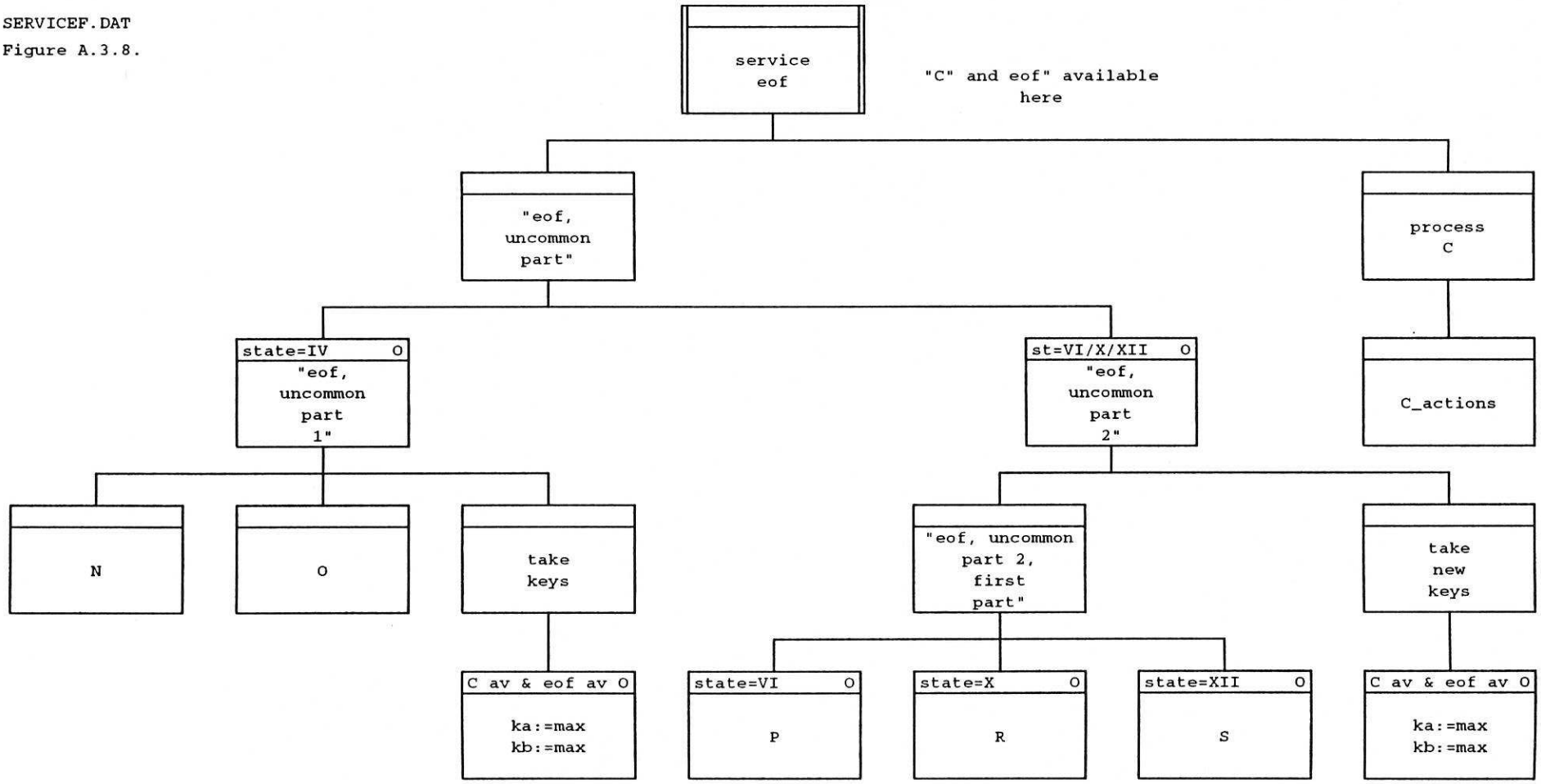
69 - Figu A.3.6 - service B

SERVICEC.DAT

Figure A.3.7



SERVICEF.DAT
Figure A.3.8.

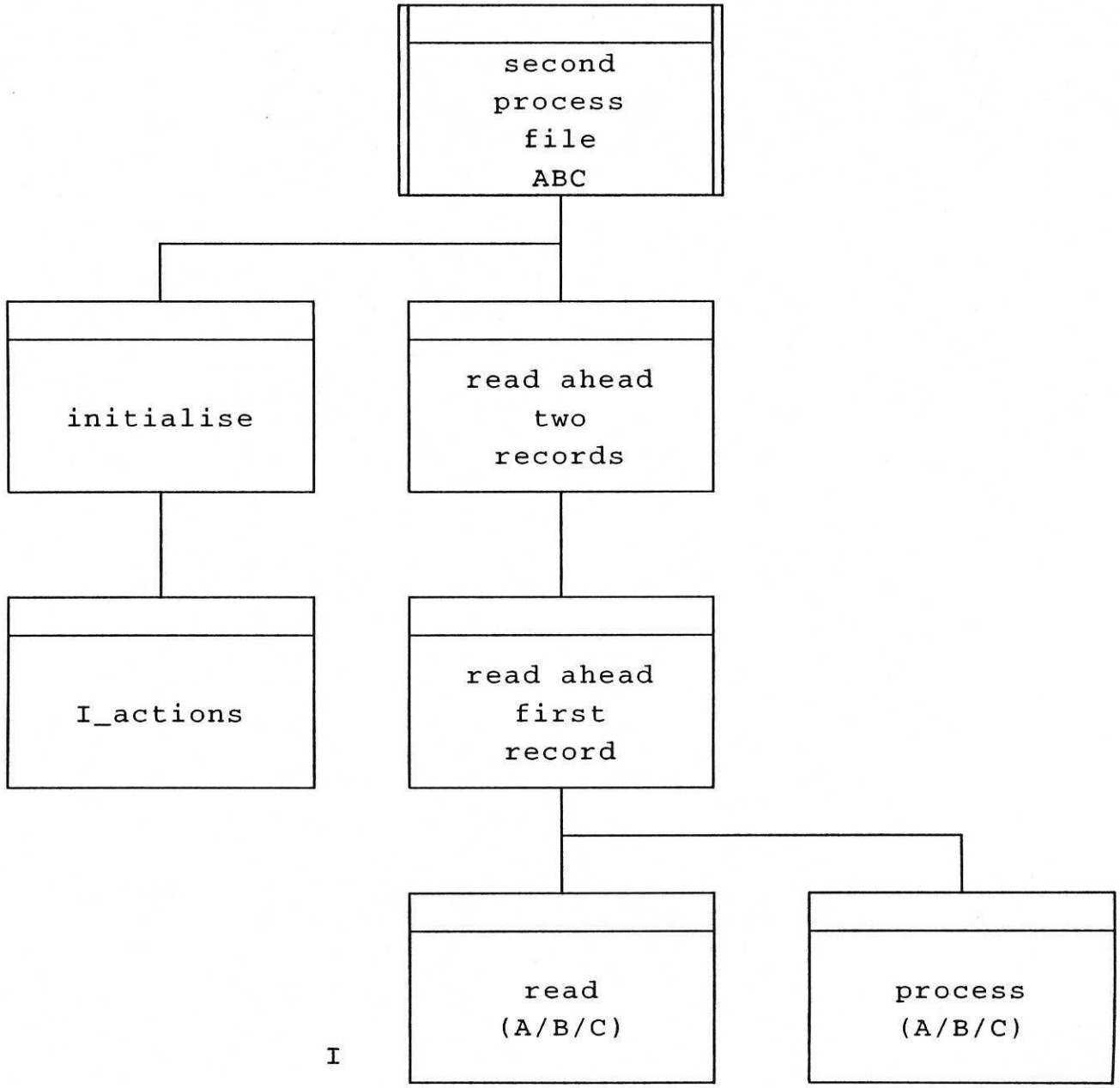


Example 4. Transformation of an OOP object into a JSD process

Figures A.4.1 through A.4.15 show, stage by stage, how a process diagram can be build up starting from its equivalent object's service diagrams. The procedure to follow is fully explained in section 6, "Deriving process diagrams from object diagrams".

2.DAT

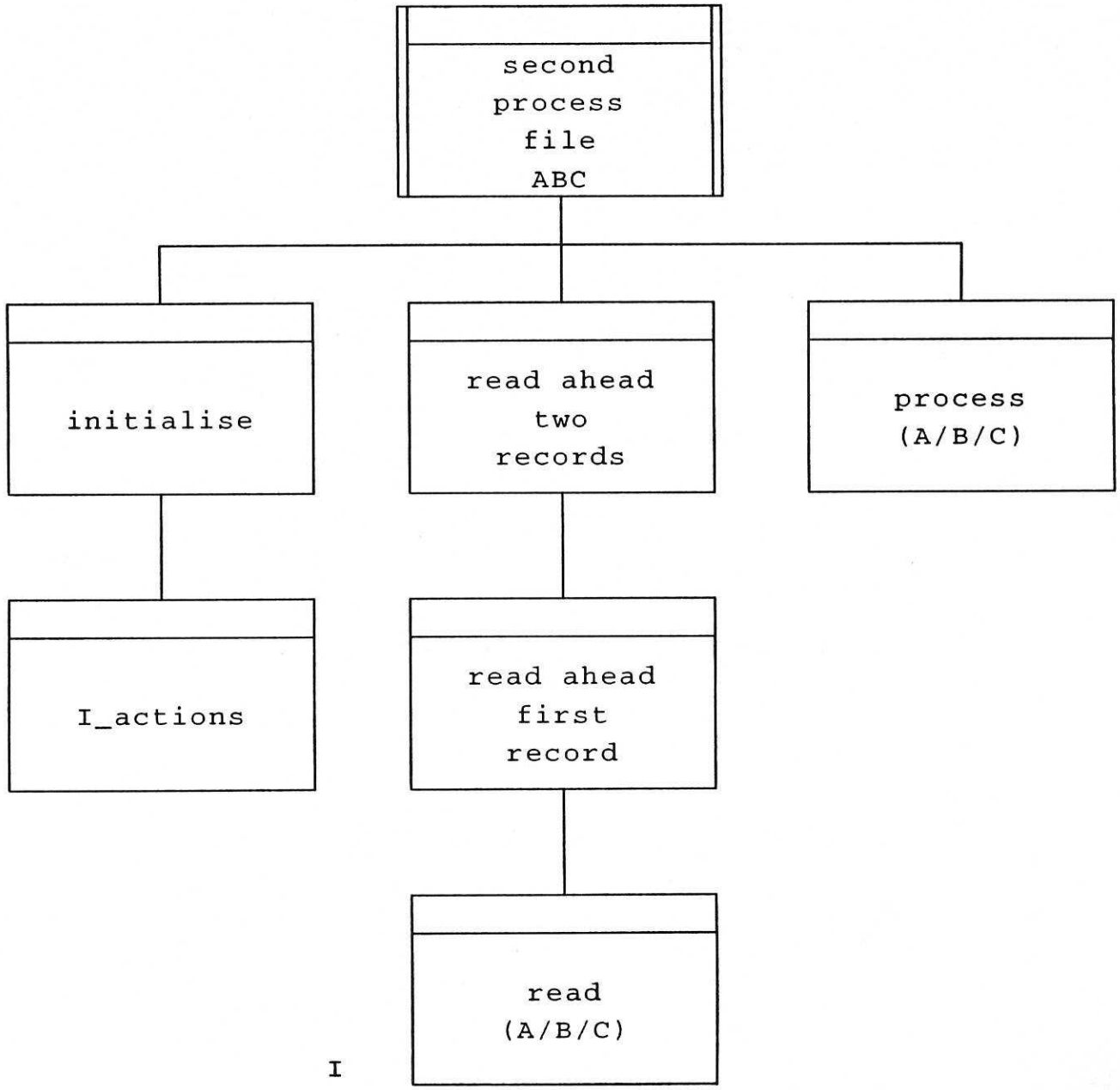
Figure A.4.1.



I

73 - Figu A.4.1 - second process file ABC

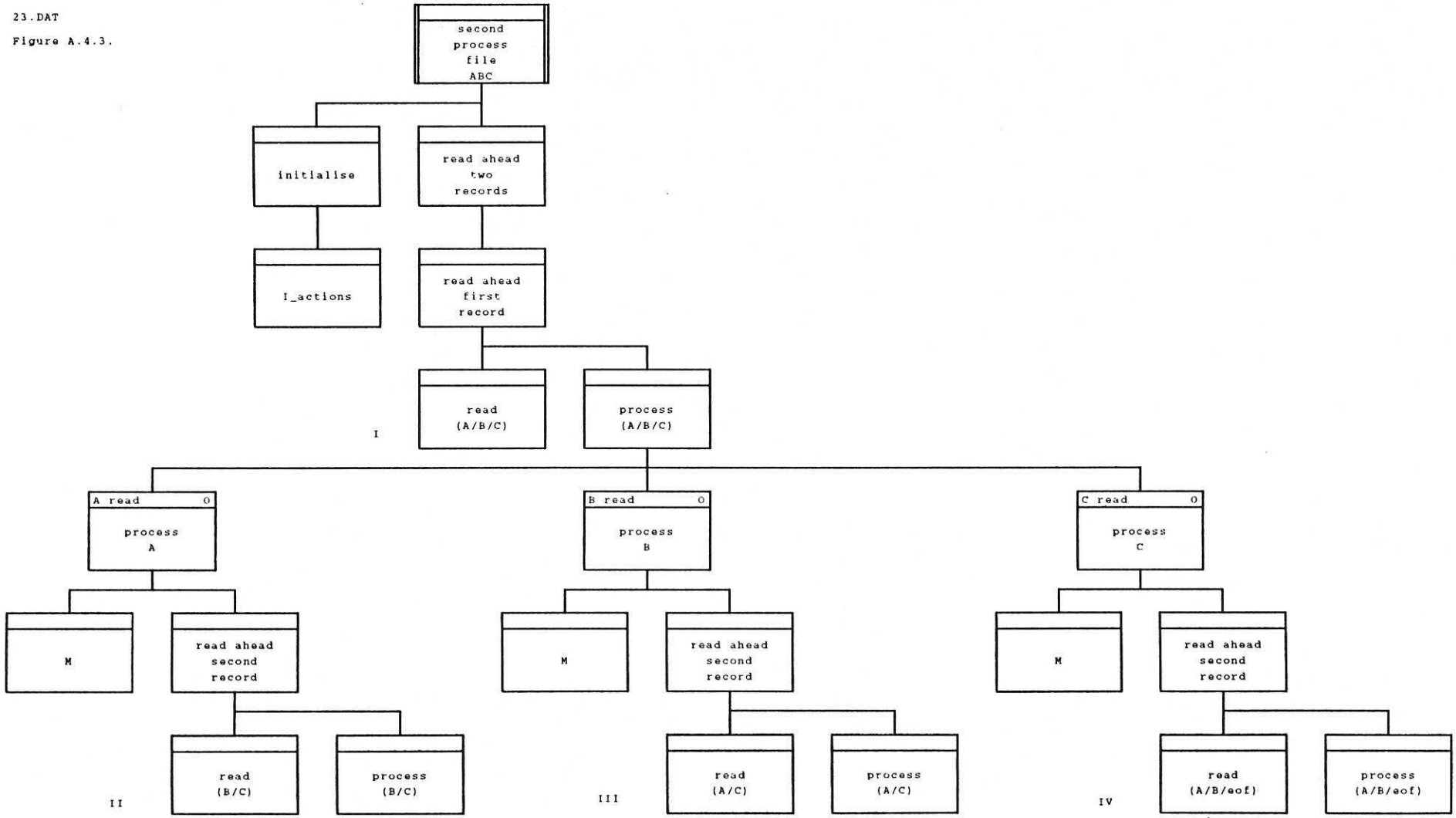
22.DAT
Figure A.4.2.



74 - Figu A.4.2 - second process file ABC

23.DAT

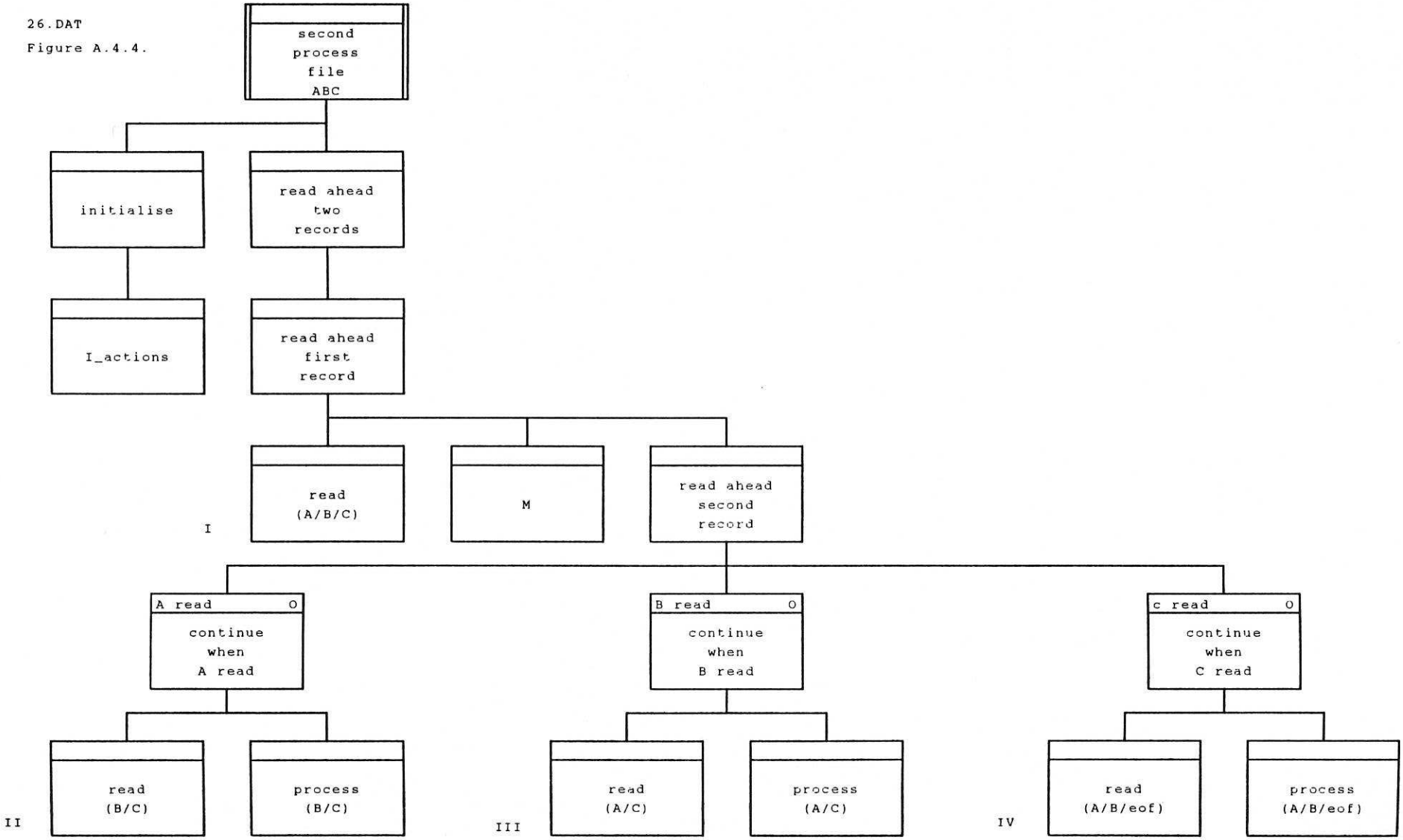
Figure A.4.3.



75 - Figu A.4.3 - second process file ABC

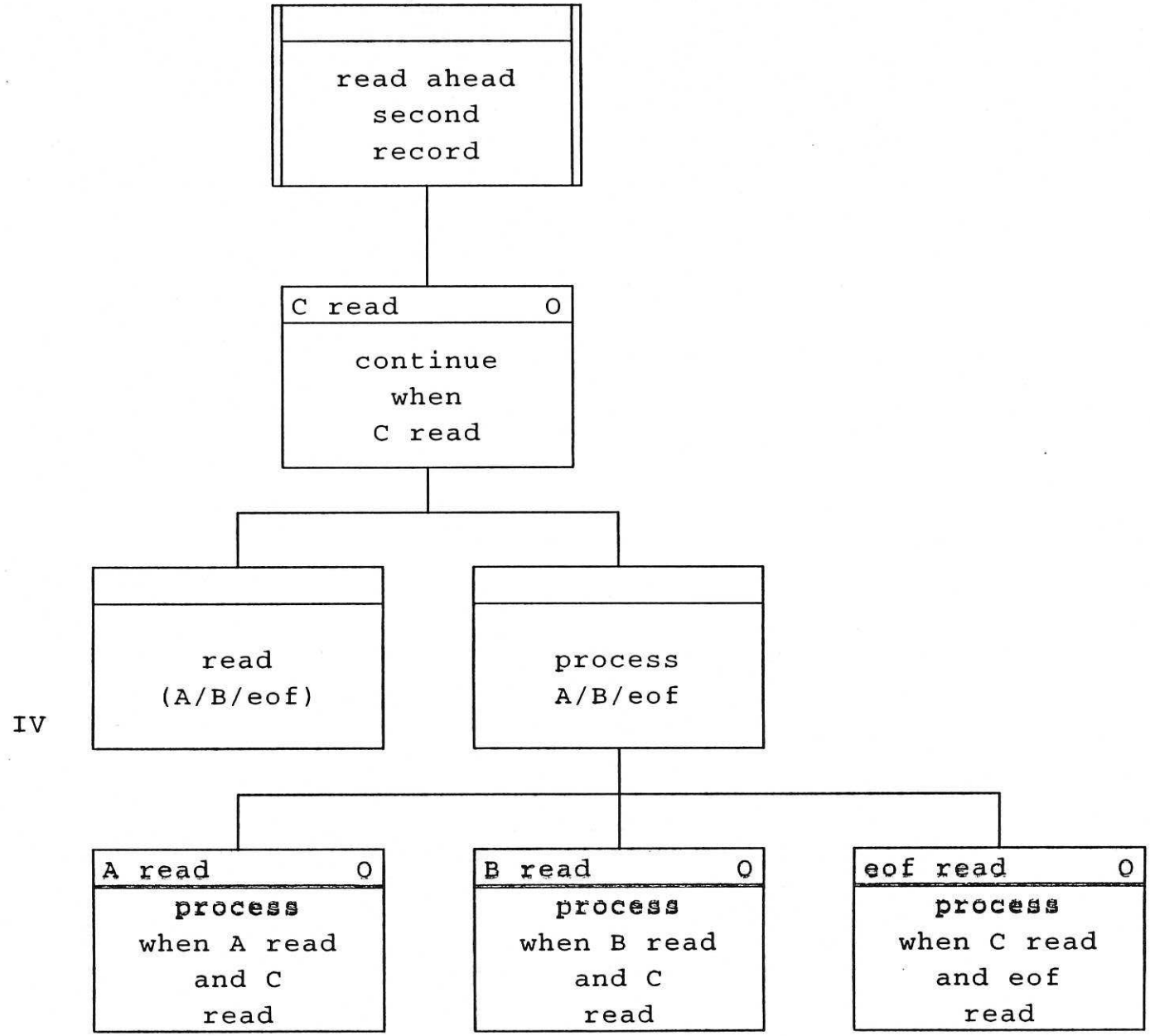
26.DAT

Figure A.4.4.



76 - Figu A.4.4 - second process file ABC

3.DAT
Figure A.4.5.

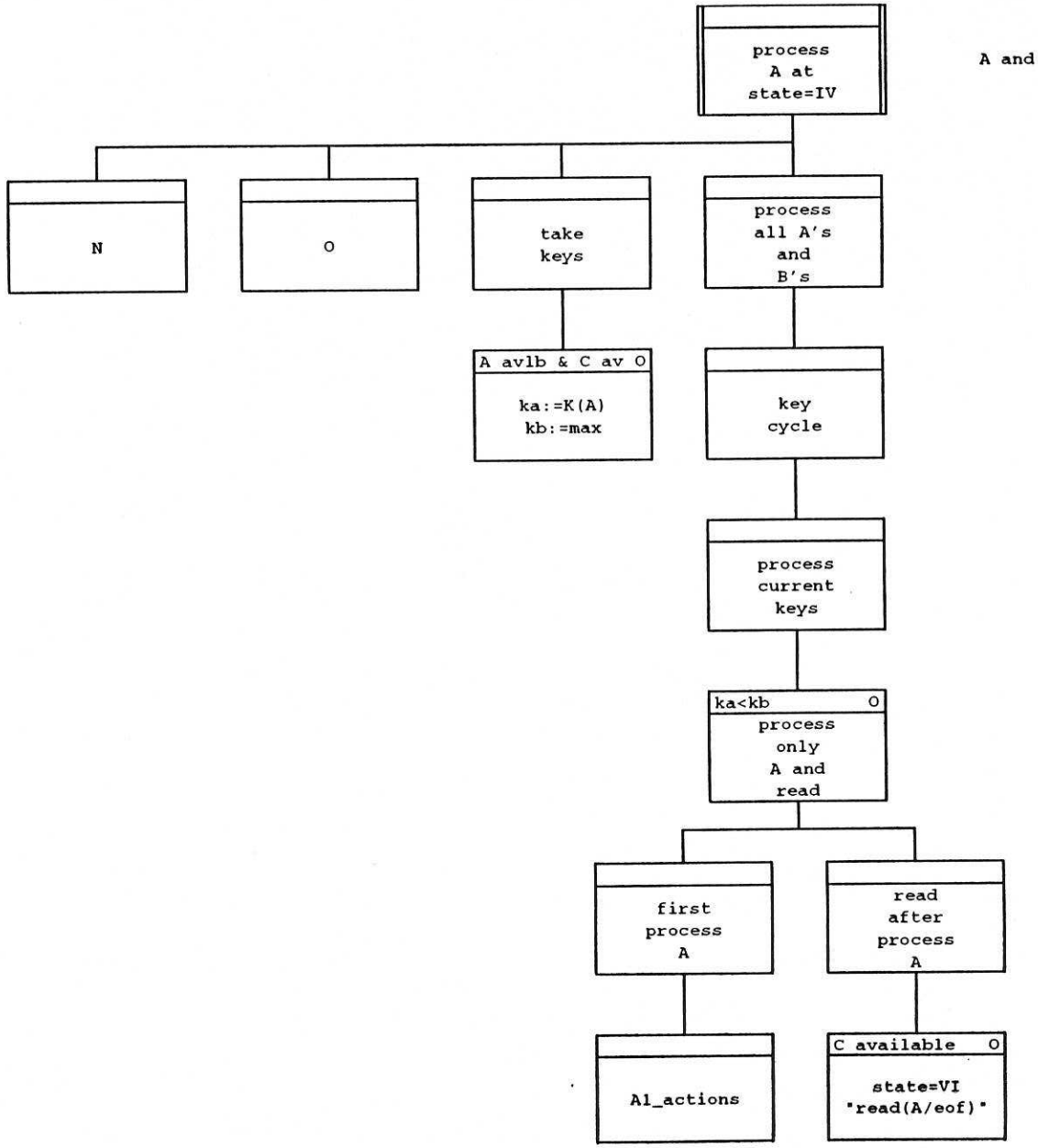


IV

77 - Figu A.4.5 - read ahead second record

4.DAT

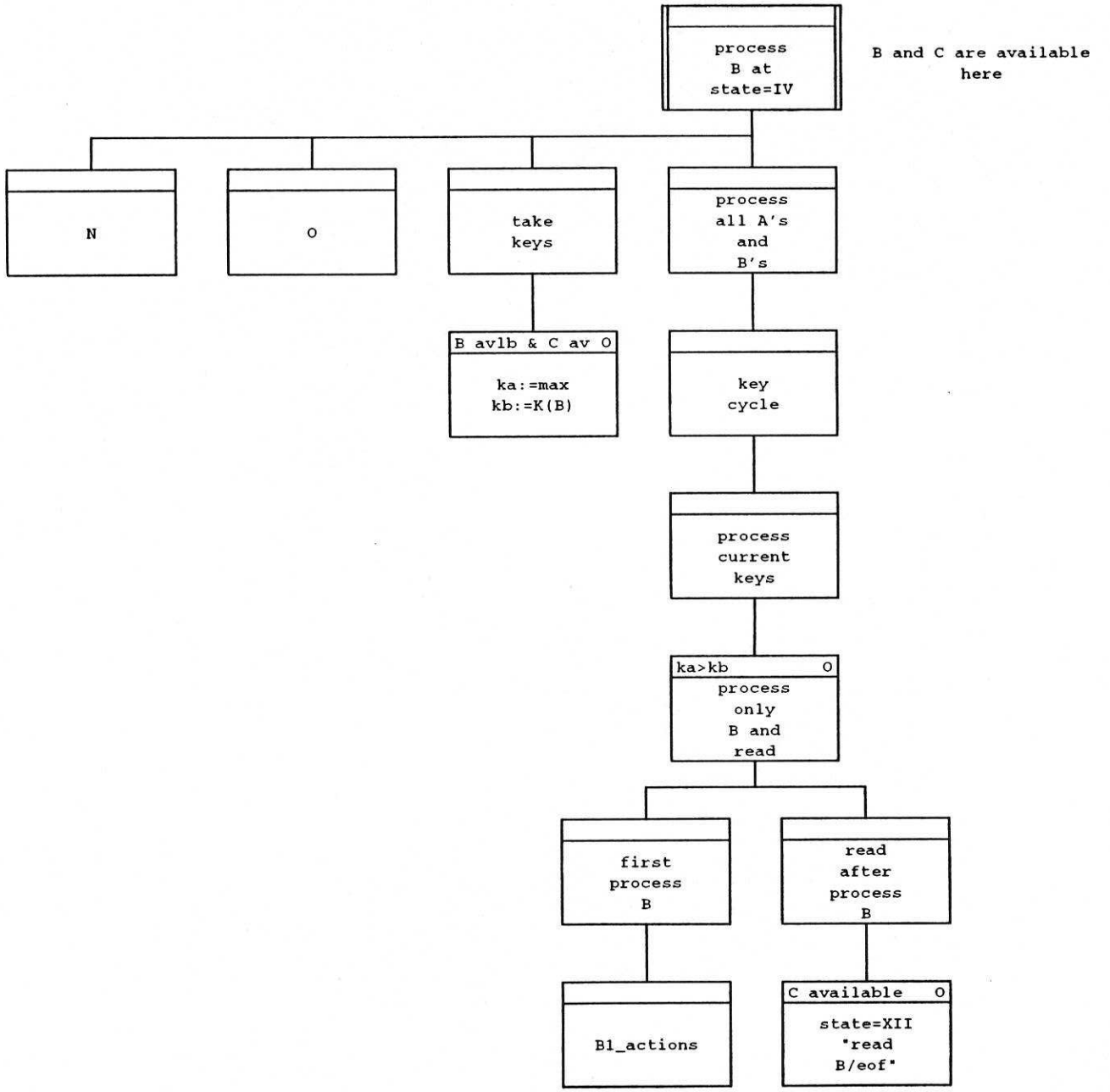
Figure A.4.6.



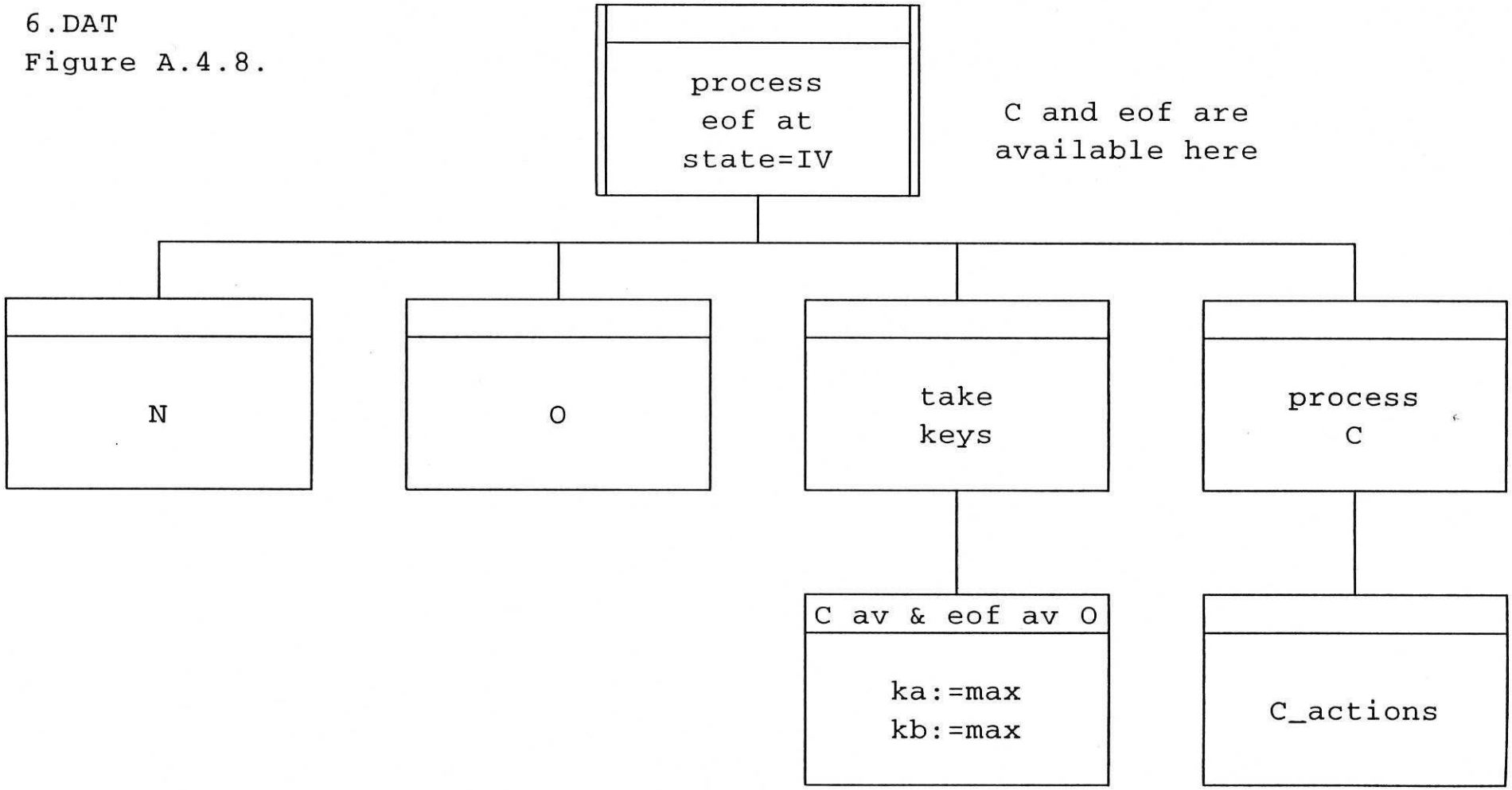
78 - Figu A.4.6 - process A at state=IV

5.DAT

Figure A.4.7.

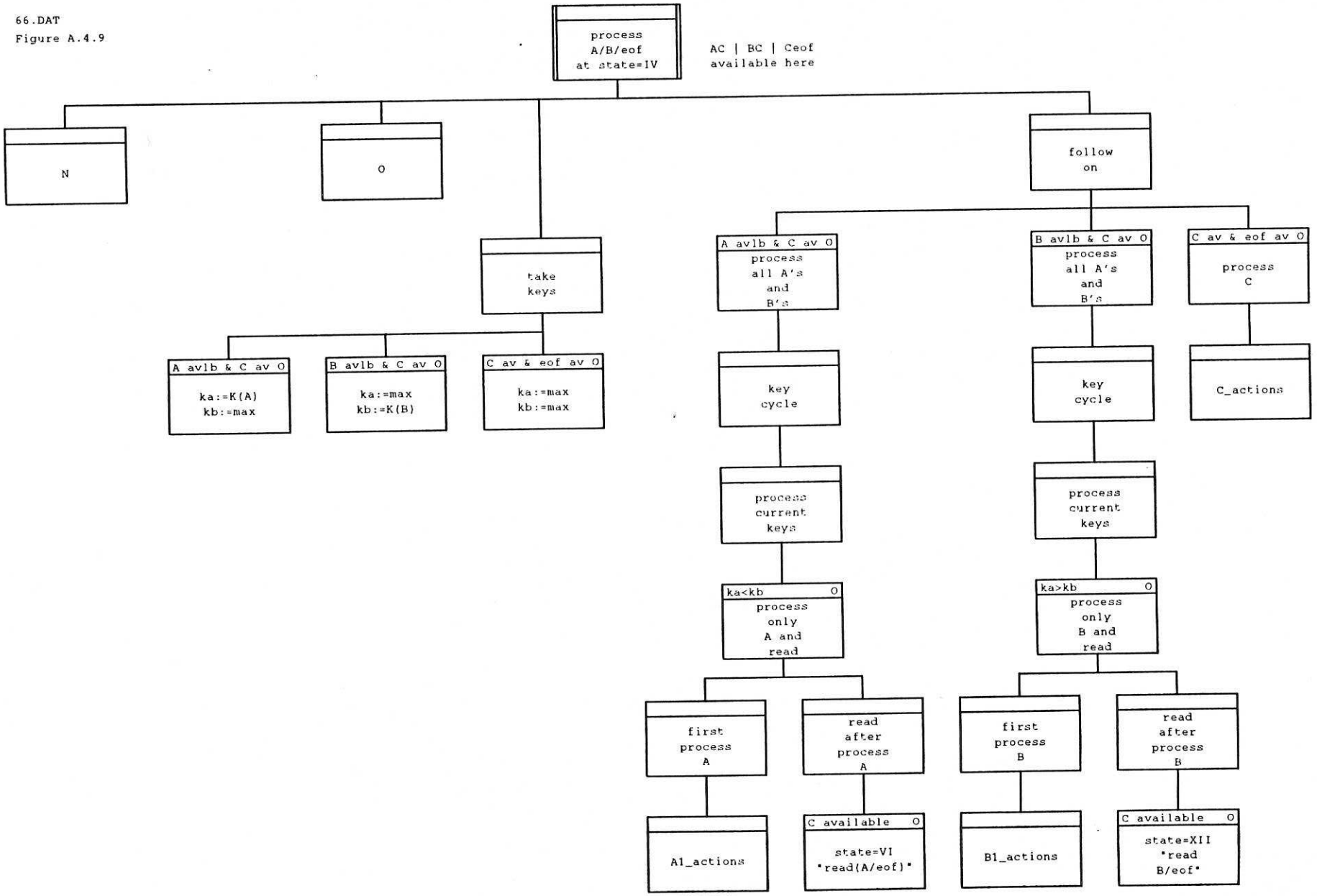


6.DAT
Figure A.4.8.



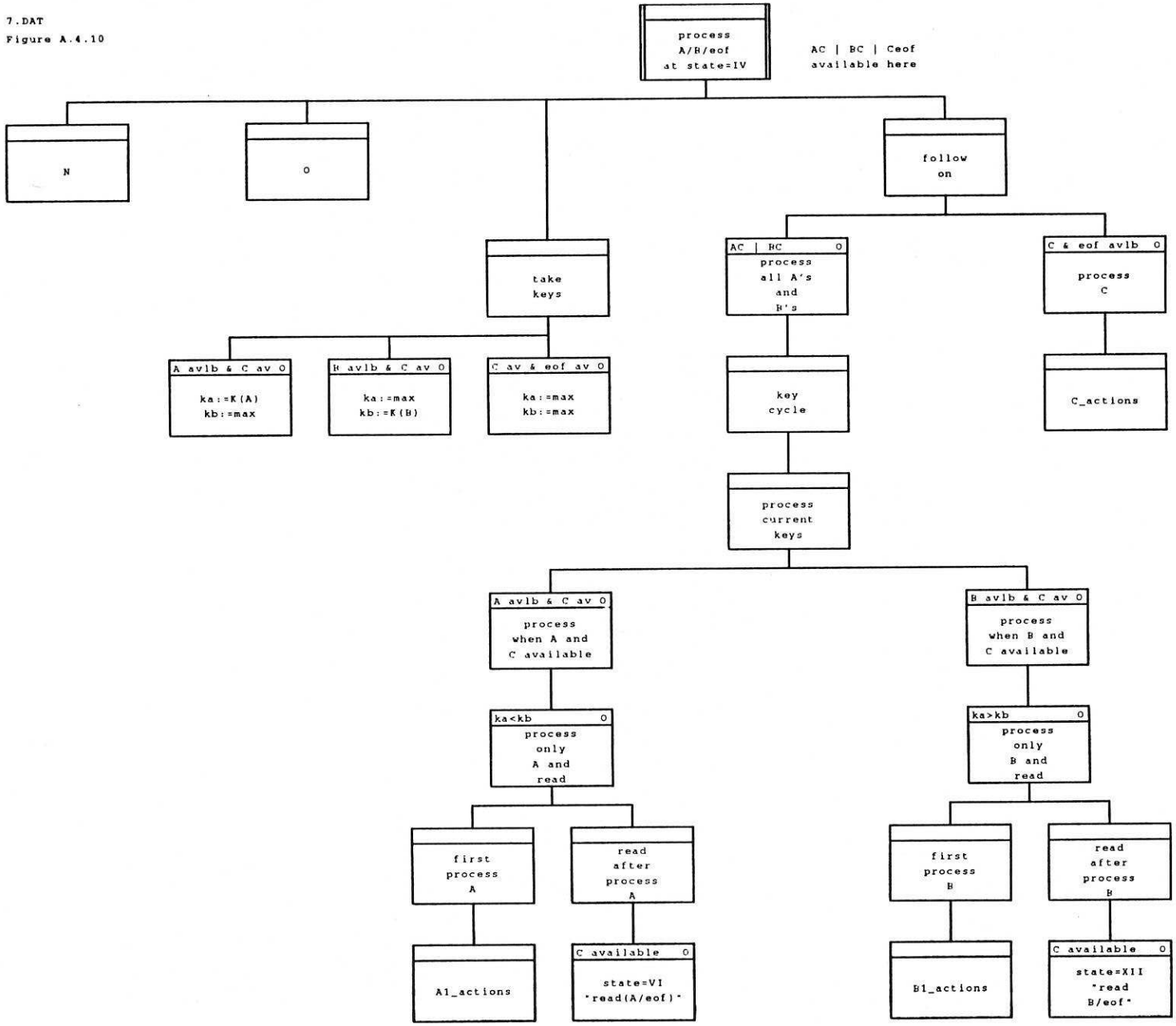
80 - Figu A.4.8 - process eof at state=IV

66.DAT
Figure A.4.9



81 - Figu A.4.9 - process A/B/eof at state=IV

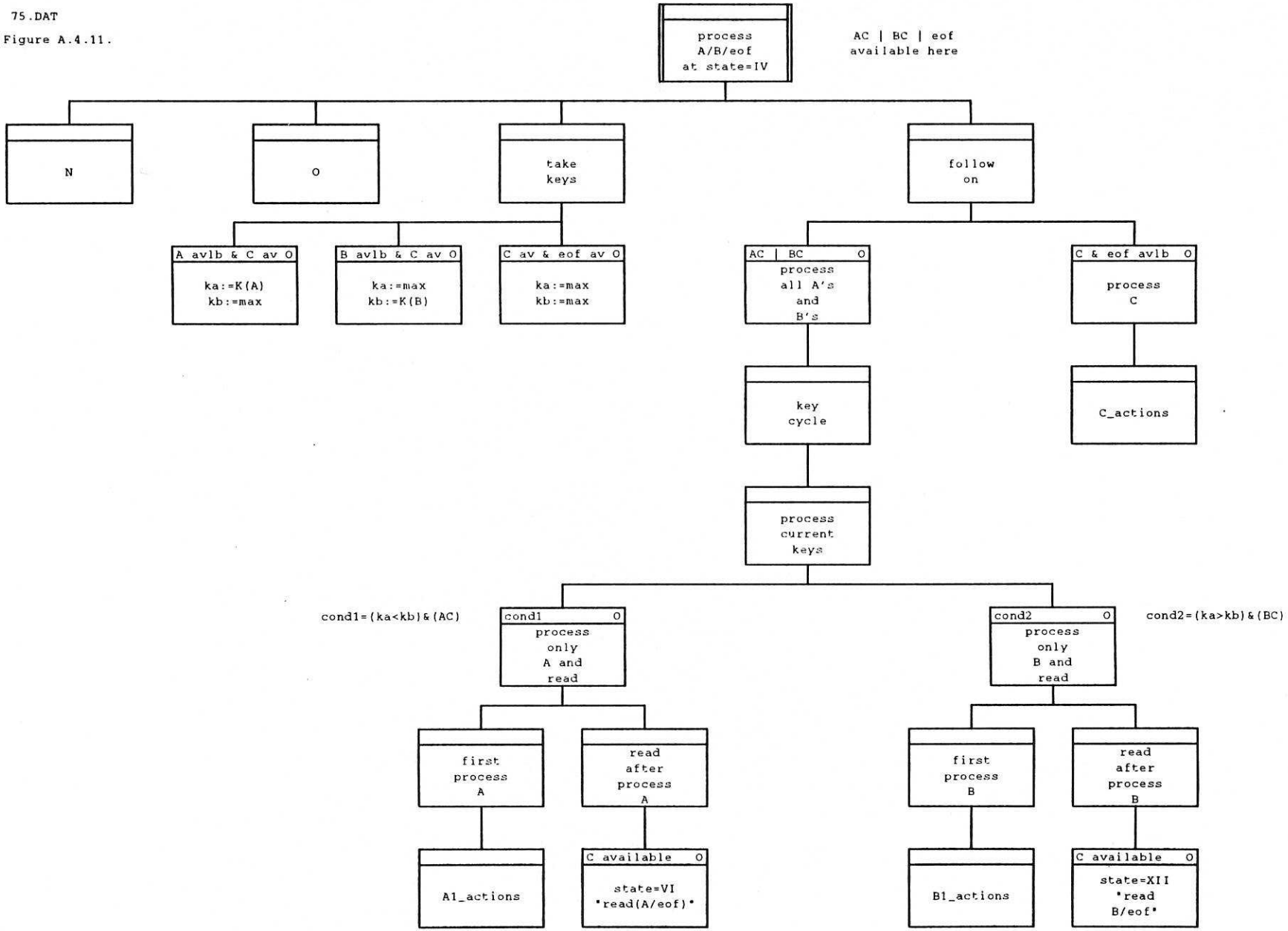
7.DAT
Figure A.4.10



82 - Fig A.4.10 - process A/B/eof at state=IV

75.DAT

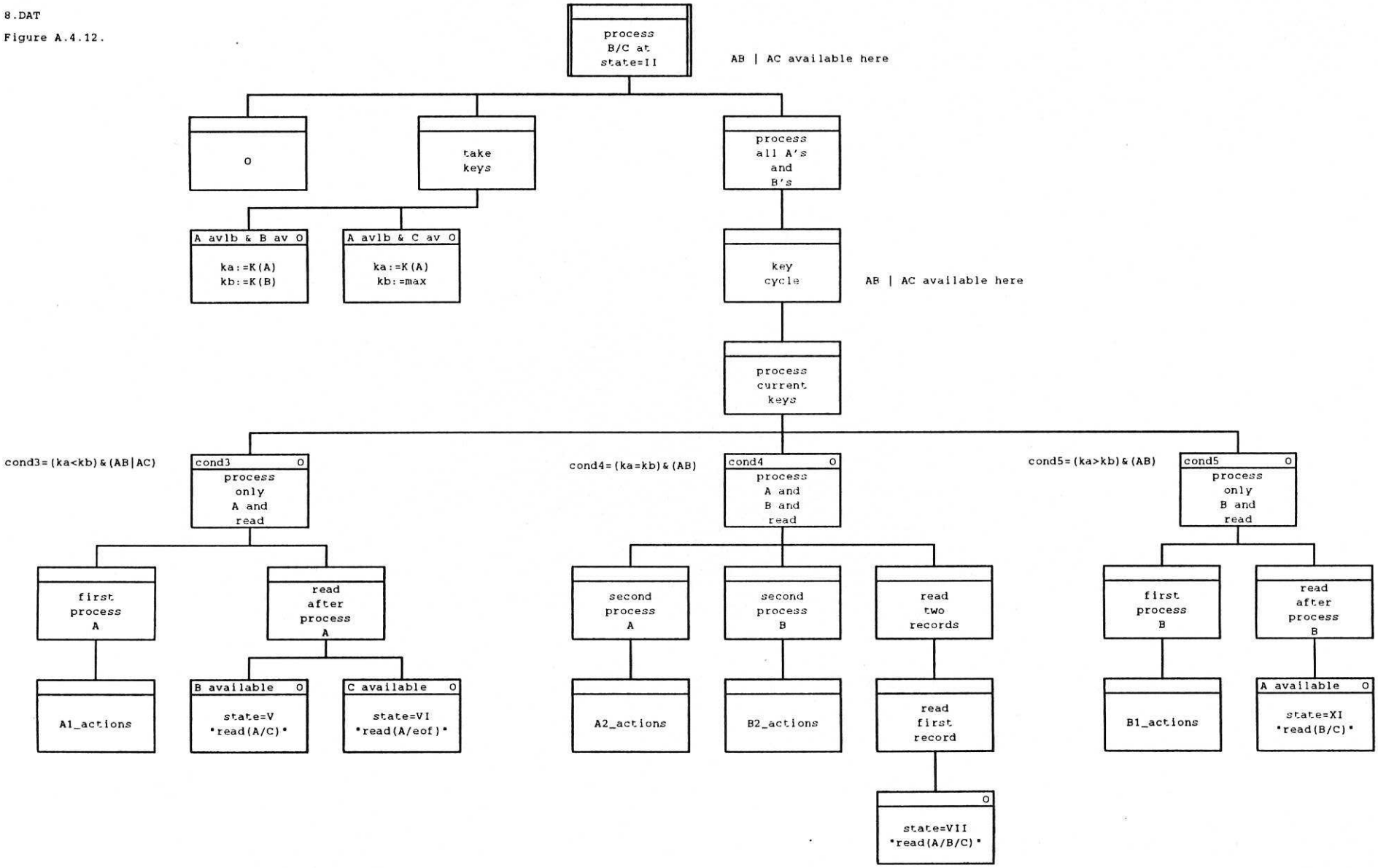
Figure A.4.11.



83 - Fig A.4.11 - process A/B/eof at state=IV

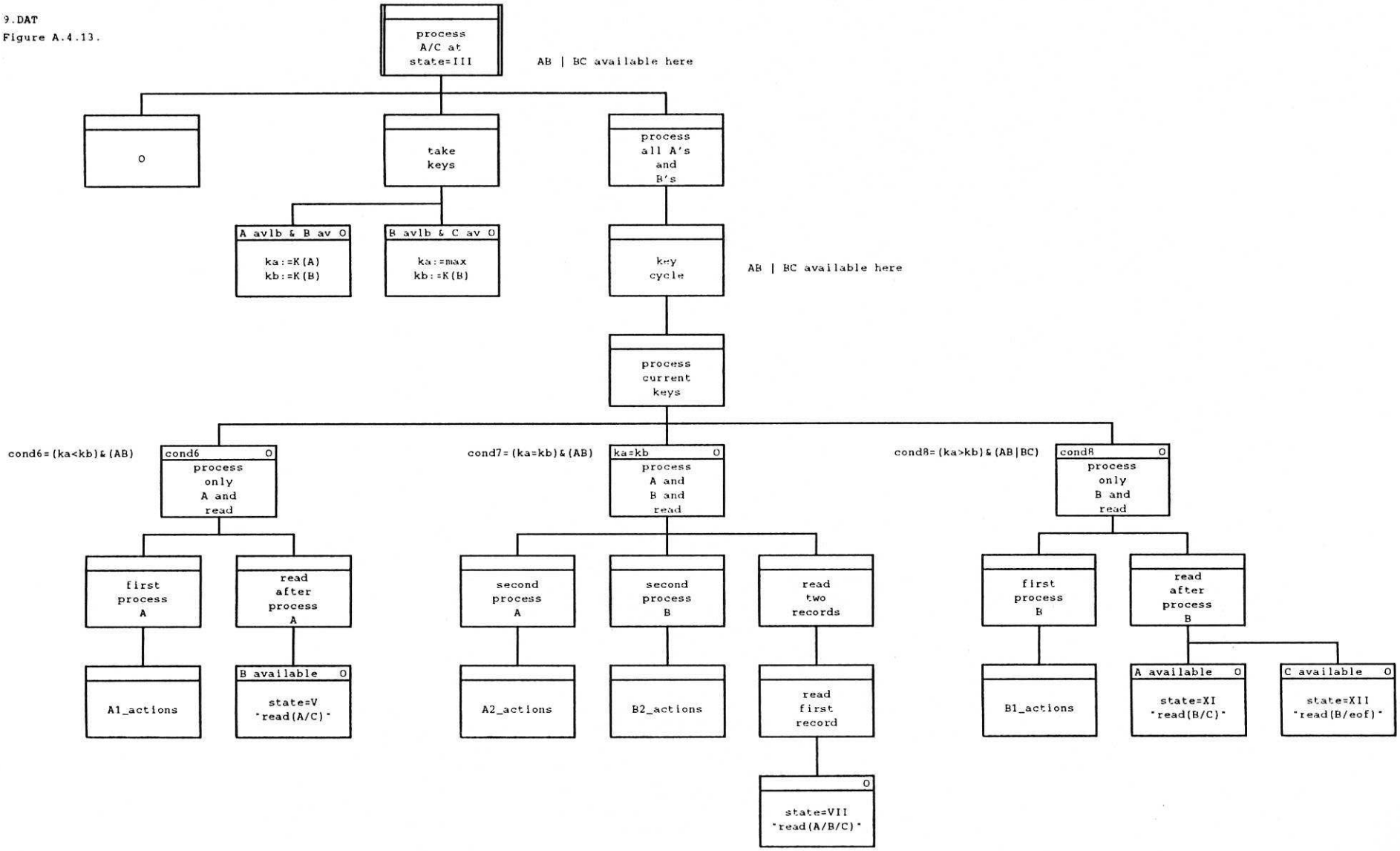
8.DAT

Figure A.4.12.

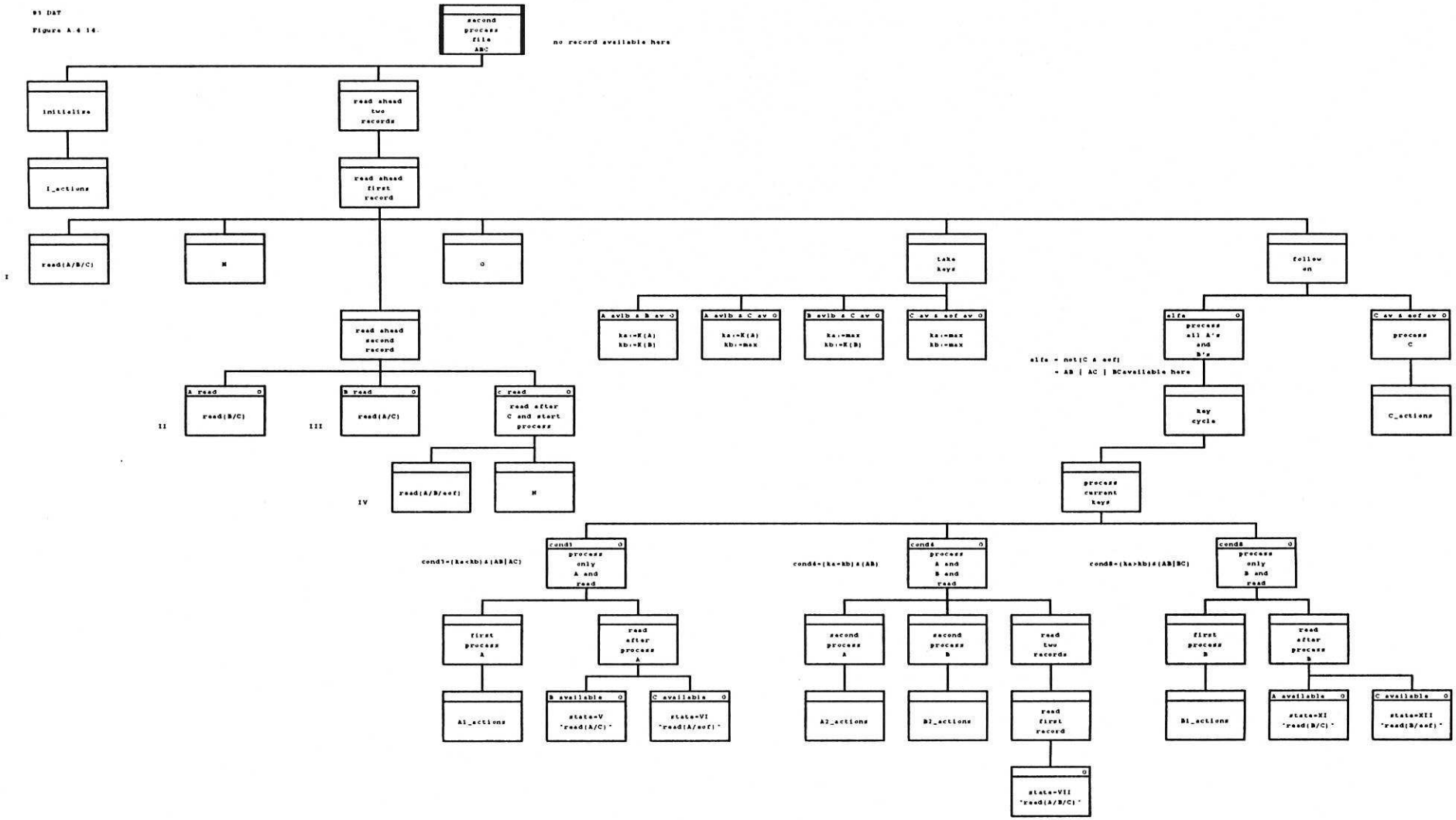


84 - Fig A.4.12 - process B/C at state=II

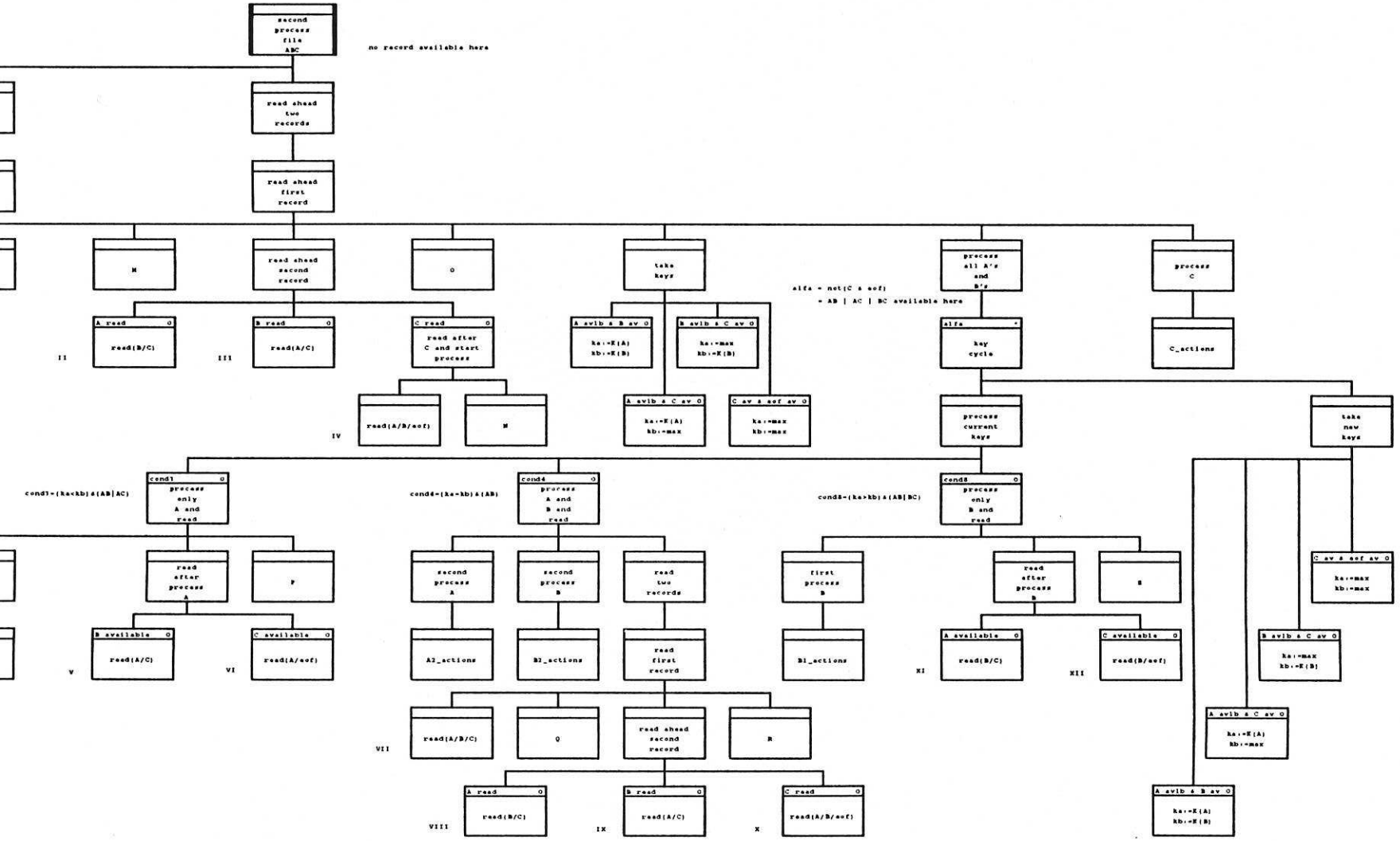
9.DAT
Figure A.4.13.



85 - Fig A.4.13 - process A/C at state=III



86 - Fig A.4.14 - second process file ABC



..15 - second process file ABC

END