# Message Flow Analysis

Oscar Marius Nierstrasz

CSRI Tech. Report #165

Department of Computer Science
University of Toronto

A thesis submitted to the Department of Computer Science and the School of Graduate Studies in conformity with the requirements for the degree of Doctor in Philosophy at the University of Toronto.

# Abstract

A message management system enables its users to automatically process messages. Procedures associated with a workstation may scan incoming mail, perform some routine processing and possibly forward the mail. The global properties of such systems may be far from obvious when large numbers of procedures are present.

We attempt to gain insight into global behaviour by studying "message flow". We do so by partitioning message domains into state-spaces, and analyzing the state transitions effected by procedures. Message flow for messages of a given type can thus be represented by a finite automaton whose states are the message states.

The finite automata for the various message types can be "welded together" to form a Petri net that accurately captures both the message flow for individual message types and the coordination by procedures of messages of different types. The model is useful for obtaining a descriptive analysis of behaviour, and for analyzing interesting behaviour such as blocking, deadlock, "message loops" and "procedure loops".

In addition we present some techniques useful for detecting message loops and procedure loops at run time.

# Acknowledgements

As is always the case in these matters (that is, the ones that go on for years and years) there are a number of people to thank. First of all, I want to thank my supervisor, Dennis Tsichritzis, whose idea this all was in the first place (though I wonder if this is what he had in mind back in 1979). Dennis' insights (usually) kept me on track, and helped keep the work interesting over the years.

I also want to thank Professors Mendelzon, Sevcik, Lochovsky, Holt, Hehner, Leon-Garcia and my external examiner, Professor Alan Shaw, for their thorough reading of my thesis (and its various incarnations). In the same breath (but a different sentence) I must thank Panos Economopoulos, who went through the nastier parts of my thesis with a fine tooth-comb long after it had stopped making sense to me.

I thank, too, the revered and respected institutions of the National Science and Engineering Research Council, the Ontario Graduate Scholarship fund, and the University of Toronto, without whose financial assistance I would be a poorer man, or possibly shy one or two degrees.

There are many other people, such as Brian Nixon, John Hogg, Kent Laver, Pat Martin, whose conversation would help to keep my mind sharp while the glowing phosphors connected to the computer would strive to deaden it.

And, of course I must thank Mom & Dad for the free rein.

# Table of Contents

# 1. Introduction

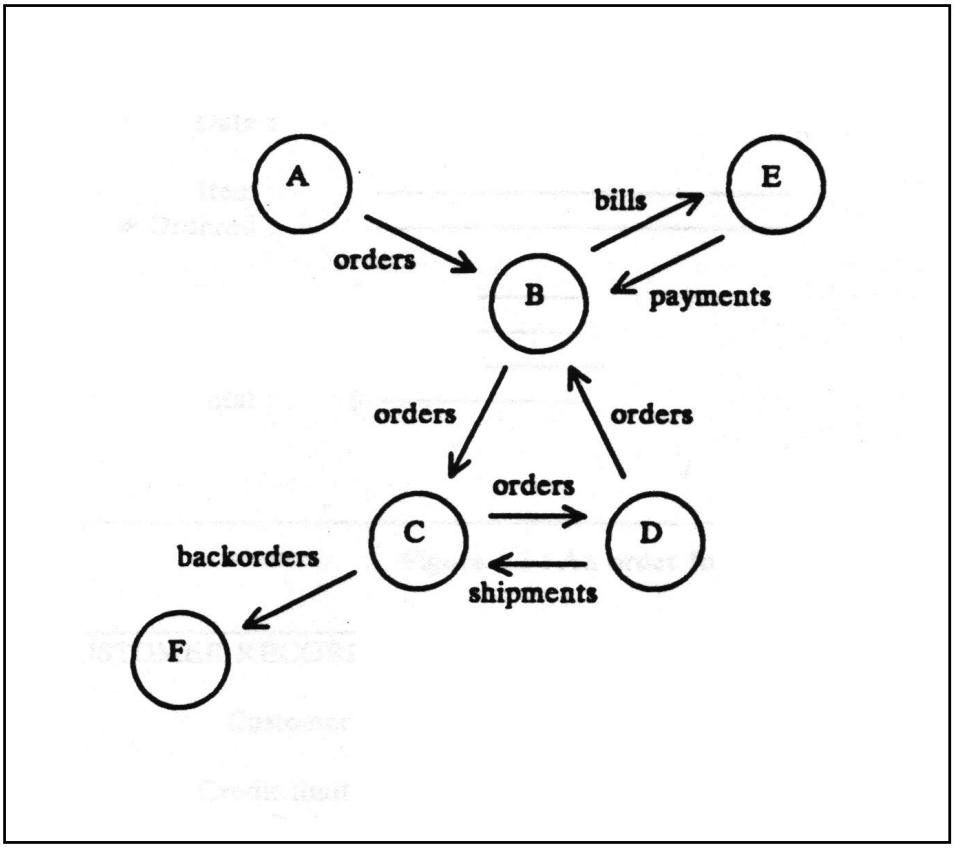
The growing interest in "Office Automation" and in naive-user programming suggests some interesting problems. In an *Office Information System* electronic documents take the place of the familiar paper documents (such as forms, letters, records and so on). As in a real office, these objects may change hands frequently. When certain activities are automated, the flow of electronic documents (i.e. *messages*) from location to location can become quite complicated. As long as these activities are well-understood and do not change frequently there may not be any difficulties. The literature suggests, however, that automation in offices will be introduced gradually, and will be subject to frequent change. The consequences of altering or adding automatic procedures that manipulate messages may not be so obvious to those who implement them. Apparently innocuous changes may affect message flow in subtle ways that can cause an existing automatic procedure to fail. In this thesis we address these problems by developing techniques for capturing the existing message flow and understanding how message flow and automation of activities affect one another.

### 1.1. Setting

Chapter 2 provides an overview to the various approaches to office automation that have been implemented. Terms such as "Office Information System" are defined. We shall introduce here the ideas that are crucial to this thesis.

First of all, we are interested in office information systems that are superficially very similar to real offices. We have a collection of *workstations* ("stations", for short) that are the logical equivalent of desks. Users communicate with each other using electronic documents or *messages* instead of paper documents. Other familiar objects may also have their counterparts in a computerized office system (bulletin boards, calculators, calendars and so on). By "simulating" a real office with the computerized system, the task of computerization is simplified and the likelihood of acceptance by office workers is increased (see chapter 2). If naive-user programming is to work, then electronic objects should have immediately recognizable counterparts to familiar physical objects, and the operations we normally perform on the real objects should translate naturally into operations on the electronic ones.

In figure 1.1 we see a small collection of workstations. The circles represent workstations and the arrows represent flow of information. *A* takes orders and sends the order forms to *B*. *B* maintains the customer records. He checks the orders from *A* to see if the customers' credit ratings are in order. If all is well, the orders are sent on to *C*. *C* maintains inventory records. If there are not enough items in stock to fill an order, *C* generates a backorder and sends a memo to F who is responsible for manufacturing. Orders that can be filled are sent to *D*, where shipping is handled. When new shipments arrive, *D* notifies *C* so the inventory can be updated. When orders are filled by *D*, the order forms are returned to *B*. *B* notifies *E* of outstanding accounts, and *E* handles the mailing of bills. When payments arrive, *E* notifies *B* so the customer records can reflect the payments.




**A**: takes orders
**B**: maintains customer records
**C**: maintains inventory records
**D**: shipping
**E**: handles bills and payments
**F**: manufacturing

**Figure 1.1** : Communicating office workstations

### 1.1.1. Messages

The static objects in such a system are electronic documents containing the information that we would normally find on paper documents. They resemble our intuitive notion of a message in that they can be sent from workstation to workstation, but in this setting they may have other constraints. Messages in an office information system may be required to continue to exist after they have been received -- documents in offices often change many hands, possibly residing at a location for a long period of time before being passed on. Furthermore, many messages fall into well-defined groups or "types". Forms and records are highly structured -- a collection of them resembles a relational database. Questions about forms can resemble database queries ("tell me what customers owe us more than a thousand dollars"). In figures 1.2 and 1.3 we see examples of structured messages. They are the order form and customer record templates of our hypothetical office.

If messages are to correspond to documents, it is useful to have unique identifiers to distinguish them. We shall assume that all messages have a unique identity. Copies can therefore be distinguished from originals. (There are many variations on the notion of a "copy": copies may be "read-only", guaranteed to be a facsimile as of the time of copying; they may be "read-only" and guaranteed to *always* represent the current version of the original -- possibly tricky to implement; or they may be writeable, gaining

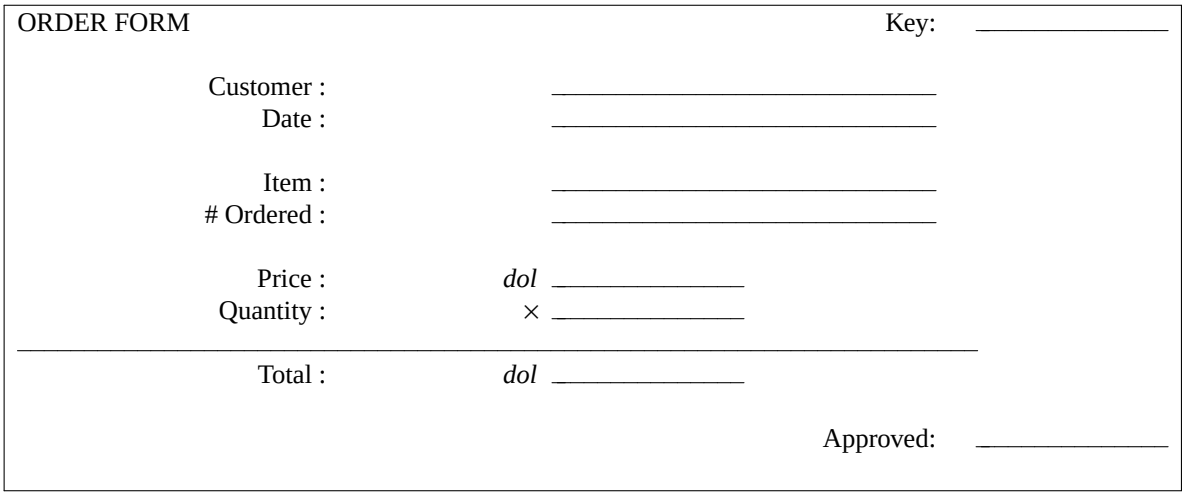



**Figure 1.2** : An order form

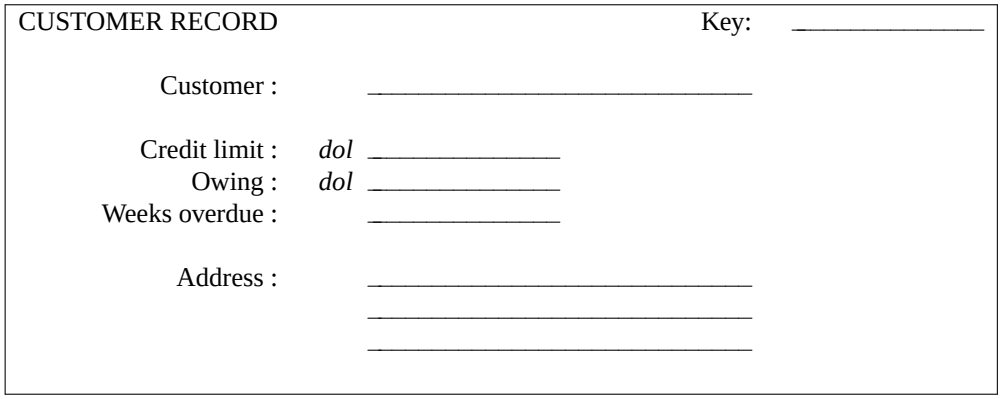



**Figure 1.3** : A customer record

independent status after their creation. Since we are concerned mainly with modifiable messages, we may pass over this issue, beyond establishing that all messages have a unique identity.) There may also be constraints on the fields of a message: some fields may be write-once, read-only, write-by-owner, and so on. Furthermore, some fields may be automatically filled, such as signature and date fields. *Automatic* fields may be calculated as a function of other fields (such as "total" or "tax" fields). *Virtual* fields are similar, but are never stored with the body of the message, being re-calculated whenever the message is displayed.

In addition to having a unique identity, we also assume that a message has a unique *location*. A location is a workstation or some mailbox owned by a workstation. (Other locations, such as archives, printers and so on, may exist, but we can consider these to be special cases of workstations.) The location of a message determines who currently has control over it. Usually this can be equated with who has the power to look at it or modify it. Exceptions are messages that may not be viewed or altered except in restricted ways by other than privileged users (such as the creator of the message). Also conceivable is a facility to allow messages to be "recalled" from a remote location. A manager may be able to track down the current location of a document such as a report or a customer record and have it mailed to him even though he is not the current owner of that location. In general, however, we assume that messages leave a station only when they are explicitly mailed by the user owning that station.

Operations on messages include creation, destruction, display, modification and mailing. In addition, since messages in this context may be a permanent record of information, we may wish to query a database of messages. Such operations as selections and joins over several messages by matching comparable fields,

for example, can be very useful. Similarly, when modifying messages, it should be possible to easily transfer data from one message to another, or to use information in one field of a message to compute or generate new information for another field.

### 1.1.2. Automation

In order to automate office activities, one must be able to recognize conditions that cause events to be triggered. Events may, in turn, cause other events to be triggered. Visible events include the arrival of messages and the creation and modification of messages. One must be able to select precisely those messages that are of interest. A trigger condition thus resembles a query ("get me a message satisfying this condition") that applies to the future rather than just the present. Since a collection of messages may be required in order to complete some activity, these conditions may potentially include joins, or matching between messages.

A simple example is mail-forwarding in figure 1.1. A procedure could automatically forward all completed order forms from *A* to *B*. Only slightly more involved is a procedure at *B* that automatically matches incoming order forms against existing customer records by the *customer* fields of the two documents. If the amount owing plus the value of the order is less than (say) half of the credit limit, and the *weeks overdue* field of the customer record is less than 4, then the procedure might automatically approve the order by forwarding it to *C*. The condition on the value of the order involves both messages, but the condition on the number of weeks overdue is a selection involving only customer records.

At *C*, a procedure could automatically match orders against inventory records. If not enough items are in stock to fill the order, the procedure could generate a backorder to be filled when the new stock arrives, and a memo to *F* that inventory is low. (The backorder is another order form that must also be matched against an inventory record.)

It is instructive to decompose activities into steps: in each step we must gather a set of resources (messages), possibly transform them in some way, and release them. New messages may be created in the process. Although an activity may consist of several steps chained together, we will concentrate on the steps themselves. The advantage of this is that we can consider the steps to be atomic -- they either succeed or fail in entirety. Multi-step activities naturally do not necessarily have this property. It is the steps that we shall speak of as "procedures", though one should keep in mind that more complex activities exist in general.

We also assume that these procedures are local to workstations. This view is very natural and consistent with the principle that computerized office systems resemble real offices: users of the system and their automated procedures only have *direct* control over the documents "belonging" to them. (We may extend this, however, by allowing the presence of local procedures at other sites that "belong" to someone else. A manager may, for example, be able to install a procedure at a worker’s station that selects and forwards certain messages back to him.) Another advantage of local procedures is that we do not have to address the problem of activities that are triggered by events that take place at several *physically* different locations. If all the "workstations" are timeshared on a single mainframe then we do not have serious problems implementing such behaviour, but it is another matter when each workstation is a separate machine on a network.

## 1.2. Global Behaviour

If automated office procedures are triggered by the receipt of mail, then it is clear that message flow and control flow are intimately connected. The current location and current value of messages determines what procedures are to be activated. Furthermore, these procedures may be "loosely connected" through the messages that they handle: a procedure that processes a message and forwards it to a new location may trigger another procedure at that site; neither procedure need have any knowledge of the other. It is (potentially) transparent whether a message arriving by mail has been sent by a manual operation or an automated one. The consequence is that the global behaviour of a system is determined *implicitly* by the collection of local procedures at all of the individual workstations -- no one needs to explicitly specify the interactions between all of the automated procedures.

If there are a lot of automated activities, then their interactions may be far from obvious. Worse yet, if procedures are frequently changed and added, then there may be unexpected consequences if the procedure interactions are not well-understood. As an example consider what would happen in figure 1.1 if *B* did not distinguish between new orders sent by *A* and filled orders sent by D. A poorly-designed automatic procedure at *B* might take filled orders from *D* and naively re-process them. A "small error" in specifying a local procedure may have repercussions that are not obvious without examining all such procedures at all locations handling messages it sends.

Another potential problem is blocking of procedures that coordinate several messages. An alteration in a procedure at one site may prevent a message from reaching a procedure at another site that is awaiting its arrival. Other messages at the second site will also be blocked.

### 1.2.1. Message flow

We can try to gain some insight into global behaviour by studying message flow. Since the flow of messages through the system determines what procedures are fired, and, conversely, the procedures determine the message flow, we can measure global behaviour in terms of the interaction between messages and procedures.

Our approach is to attempt to classify messages according to the paths they take. Two messages are in the same equivalence class if their potential paths are identical. Intuitively, the path of a message is the sequence of procedures it encounters. In addition, since procedures are conditionally triggered on the current values of their input messages, we find it convenient to think of message paths as alternating sequences of procedures and values. Since procedures typically select some messages of a given type (matching certain conditions) and ignore others, the equivalence classes of messages following similar paths can be expected to have comparable values. If we can partition message domains into sets of message values that satisfy various combinations of these conditions, then we should be able to express the paths of our equivalence classes as alternating sequences of procedures and blocks of our partition.

Let us consider the following example. In figure 1.4 we see a graph model of the procedures described earlier. In fact, it is a Petri net representation of those procedures (see appendix C for a discussion of Petri nets). Nodes in the graph are bars (called *transitions*) and circles (called *places*). The transitions, labeled $t_1$ to $t_6$ represent procedures. The places represent the messages handled by the procedures. They are labeled with a letter representing the message type and a subscript. Places labeled *o* are order forms, *c*, customer records, *i*, inventory records, *l*, low-inventory memos, and *b*, bills. Procedure $t_1$ at station *A* is the procedure used to generate and forward order forms to *B*. $t_2$ checks the customer's credit rating and forwards the order form to C. $t_3$ generates back-orders for low-inventory items (arc to $o_2$) and sends low-inventory memos to F (at $l_1$). The back-orders must again be matched against inventory records (when the supply is replenished). $t_4$ handles orders that can be filled immediately. $t_5$ forwards order forms back to *B* after the orders have been filled. $t_6$ updates the customer record, archives the order form and generates a bill to be handled by *E*. (To keep the example manageable, we have left out procedures handling customers with poor credit ratings, arrivals of new shipments, and customers' payments.)

Figure 1.4 differs slightly from an ordinary Petri net in that there is a correspondence between input and output places. In a Petri net, inputs (represented by dots, or *tokens* within the places) are consumed, and outputs are produced. In our case, an input *from* $o_2$ goes *to* $o_3$. We may highlight this correspondence by concentrating on a single message type at a time. In figure 1.5 we see a new Petri net obtained by deleting all places representing messages other than order forms, and all arcs directed to or from the deleted places. In addition, we add a new place, $\alpha$, representing the creation of order forms. The resulting Petri net has the property that every transition has precisely one input place and one output place. Such a Petri net can be viewed as a finite state automaton. To see this, simply eliminate the bars from the figure and use the Petri net transition labels as the state transition labels. The places of the Petri net become states of the automaton. $\alpha$ is the initial state of the automaton. (Note that $t_3$ had to be split into two transitions since it had the unusual property of outputting two messages of the same type.) The sequences of possible transition firings for the Petri net can therefore be described by a regular language.

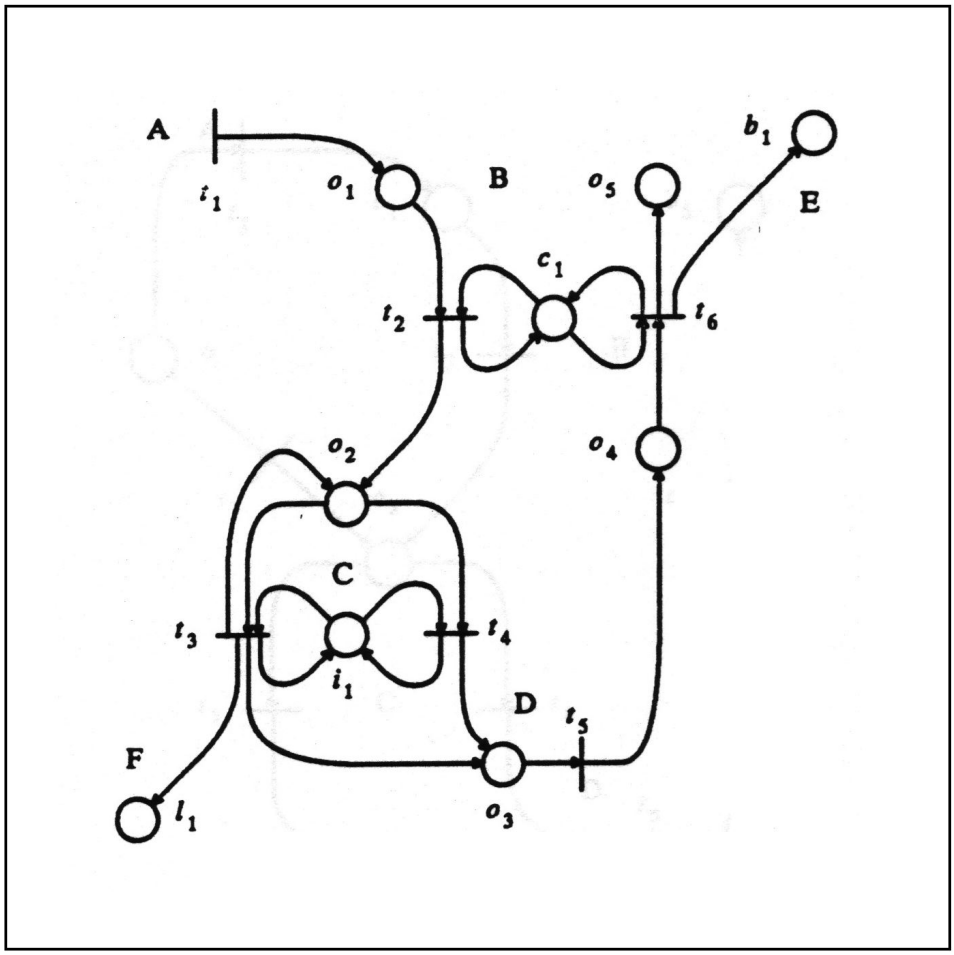



**Figure 1.4** : Petri net of procedure interaction

By this reduction of the Petri net into finite automata (one per message type), the message flow inherent in the system becomes apparent. We can now see that there are two equivalence classes of order forms: one created by $t_1$ and the other created by $t_3$. (Once any given order form reaches state $o_2$ we do not know whether $t_3$ or $t_4$ will fire since we cannot predict the value of the matching inventory records. All order forms are thus "equivalent" in the sense that we cannot deduce anything more about their future path.) The regular expressions for the two classes are:

$$\alpha\ t_1\ o_1\ t_2\ o_2\ (t_3 + t_4)\ o_3\ t_5\ o_4\ t_6\ o_5$$

and

$$\alpha\ t_3\ o_2\ (t_3 + t_4)\ o_3\ t_5\ o_4\ t_6\ o_5$$

where "+" denotes alternation (see chapter 4).

In this example the states of the automaton are distinguished by the location of the messages. $t_3$ and $t_4$ are triggered by the presence of order forms arriving from *B*. It is important to be able to tell who sent a message -- to this end we find it convenient to think of separate mailboxes for mail arriving from different sources. We may thus encode the source of a message in its location. (If this seems slightly odd, one may equivalently think of messages residing in a mailbox as having an additional *sender* field, which is set to null when it is removed from the mailbox. Incorporating this information into a single *location* field seems somewhat cleaner, however.) Messages at $o_1$ arrive from *A*, messages at $o_4$ arrive from *D* and messages at $o_5$ reside locally at *B*. If we were unable to distinguish these three, then they would collapse into a single

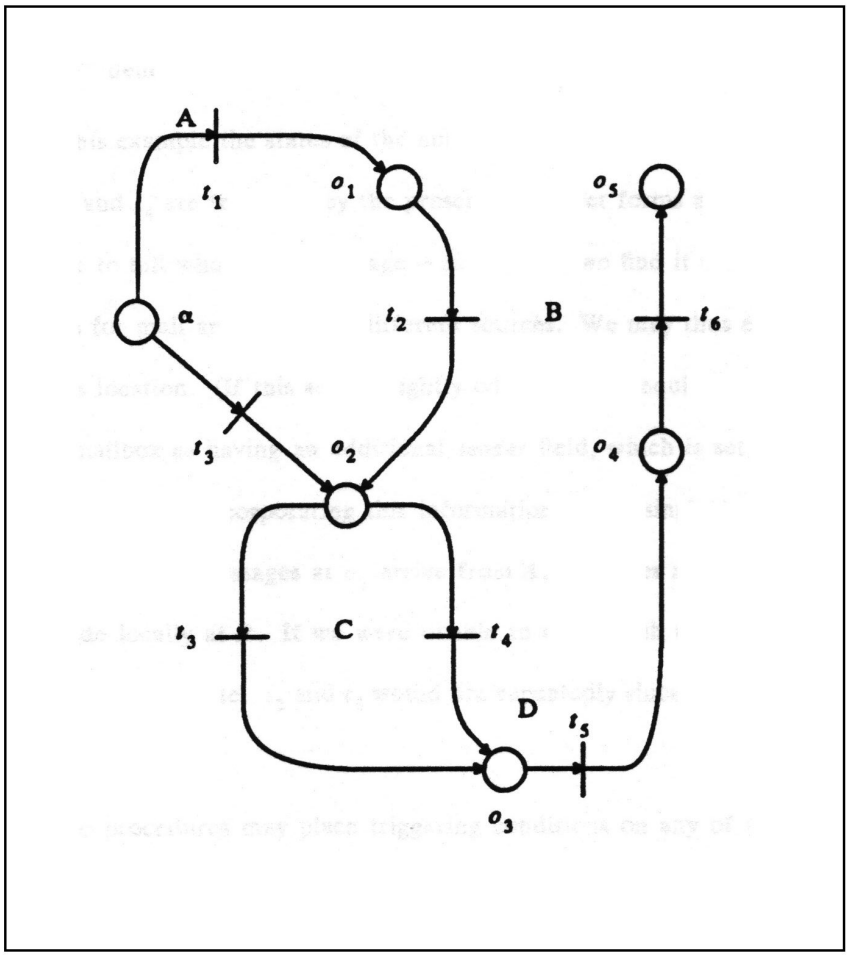



**Figure 1.5** : Finite automaton of message flow

state. $t_2$ and $t_6$ would fire repeatedly since messages would always return to that state.

Since procedures may place triggering conditions on any of the fields of a message, the states of the automaton can potentially correspond to conditions on fields other than the location. If, for example, order forms contained an *order filled* field (containing the shipment date, or some other information) and a *billing date* field then $t_6$ could ignore the sender of *order* messages and select those with non-null *order filled* fields and null *billing date* fields, setting the latter the appropriate value. In chapter 4 we shall look at how to convert selection trigger conditions of procedures into *message states* (the states of our finite automata).

### 1.2.2. Evaluating message flow

A characterization of message flow is an expression of the global behaviour resulting from a given configuration of local procedures. This in itself can be very useful if these procedures are frequently changed. Message flow can also be used as an indicator of interesting or unusual situations arising from the interaction of procedures.

First of all we see what procedures are responsible for creating messages. (This information is readily available in the specification of the procedures.) We also see where messages terminate. Clearly, if a message is routed to a station that has no procedure prepared to handle it, then it can go no further. The matter is complicated, however, when there *are* procedures prepared to handle some messages of a given type, but not necessarily all of them. Of course, if messages that can't be handled never arrive anyway, then

this does not cause any problems. In order to find out whether or not this is the case, however, we may need to trace the flow of all messages of that type to see where they end up. In addition, there will undoubtedly be points where messages are expected to terminate (as in figure 1.5). We are interested in finding out if any messages may terminate unexpectedly.

Message loops may occur if messages return to a station they have visited before. If neither the value of the message itself nor those of coordinating messages are changed in the body of the loop, then the loop may be repeated indefinitely. Another possibility is that message values change but do not alter the triggering of procedures within the loop, thus preventing the loop from being exited. This could happen in figure 1.4 if procedures at *B* did not distinguish between order forms coming from *A*, those coming from *D* and those already at *B*. Places $o_1$, $o_4$ and $o_5$ would be collapsed into a single place here and also in figure 1.5. $t_2$ would not distinguish between $o_1$ and $o_4$; the inventory would steadily decrease and the customer's bill would increase, however, causing the loop to terminate when stock ran out or the credit limit was exceeded. If the values do change, as in this example, but not necessarily enough to cause the loop to be broken, then there may be an arbitrarily large number of iterations before it does terminate. Sometimes (as would be the case here) the presence of any message loop at all is an obvious error. Other times it may simply reaffirm what is expected: the inventory records handled by $t_3$ and $t_4$ are in perfectly acceptable loops.

Blocking occurs if a message arrives at a point from which it can make no progress. There are several ways this may happen. First, a message may simply reach a dead end where no procedure will handle it ($o_5$ in figure 1.5). Next, a message may be stuck at a procedure waiting for a coordinating message that will never arrive. The coordinating message may never have been created; it may have been routed (non-deterministically) so as to avoid the waiting message; there may not *be* a path to the waiting message; the coordinating message may be blocked itself. Care must be taken here to establish *how many* possible coordinating messages there may be, and where they come from. One such message may well avoid our waiting message, but another one may be on its way. Consider a given customer record at $c_1$ in figure 1.4. If no orders arrive for that customer, then that record is blocked. Orders originating at $t_3$ may never get to $o_4$ if they are blocked at $o_2$. Other orders may well be generated by $t_1$, however, so that the customer record would still not be blocked. Blocking could only occur if *no* coordinating message arrives. Deadlock is a special case of blocking in which two (or more messages) are blocked, waiting for each other at different locations.

Another situation, related to message loops, is that of procedure loops. It is possible for a loop to exist in the firing of procedures *without* any given message indefinitely repeating a segment of its path. Suppose, for example, that procedure $t_3$ in figure 1.4 fired even if there were *no* items in stock with which to even partially fill the order. Orders arriving at $o_2$ would have the quantity set to zero and be sent on to $o_3$. A new backorder for the original quantity would be generated. This, however, would create a procedure loop if the new backorder could not be distinguished from the original. $t_3$ would fire repeatedly, generating backorders, zeroing them and mailing them to *D*. (More apocryphal is the example of a procedure to automatically respond to mail messages while you are on holidays with a note telling when you will return. If two people have such procedures, and the last act of one before leaving is to broadcast a farewell message, then the two auto-mailers will enter a loop, sending notes to each other until the file system is filled or somebody notices and pulls the plug.)

Procedure loops, like message loops, are of greatest concern when they are unmoderated, and when there is no indication that the loop will terminate. (A loop is "moderated" if any of the procedures in the loop must wait before firing for some outside event, such as user input.) The Petri net model of procedure interactions can be used to detect procedure loops. This matter will be discussed in more detail in chapter 5.

Non-determinism in global behaviour can have many causes. Outright conflict may exist with two or more procedures whose triggering conditions overlap. If there are no rules to determine which procedure is to be fired, then it may be up to the system to arbitrarily choose one. Conflict may also occur in a weaker sense when it is resolved by a coordinating message. For a given order form at $o_2$ in figure 1.5, either $t_3$ or $t_4$ may fire depending on the matching inventory record. Without knowing the values of all inventory records for all time, we cannot predict which procedure will end up handling order forms. Furthermore, the

system is not, in general, closed, since user input can introduce new values and new messages (a closed office system would probably not have much relevance to the real world). The uncertainty of these new values limits our ability to predict the behaviour of the system after the values are added.

Finally, there is some artificial non-determinism introduced by our attempts to model message flow. In order to keep the model manageable we partition message domains (the sets of possible values messages may assume) into a finite number of *message states*. We can then speak of procedures causing messages to flow from one state to another. How we make the partition inevitably influences the strength of our results. Theoretically speaking, our message domains are infinite, but since messages must be represented within a computer, we know there is only a finite (though very large) number of possible values. Allowing one message state per value will clearly yield the maximum possible information about message flow, though at enormous cost. Allowing a single message state consisting of all values yields a trivial model but sacrifices all message flow information; all procedures would map messages from that single state back to that state. In chapter 4 we shall consider the options available to us in choosing the message state space. When drawing conclusions from our model we must take care that the information we seek has not been destroyed or altered by our choice of a state space. Non-determinism displayed by our model may not necessarily correspond to non-determinism in the systems we are studying.

### 1.3. Outline

In this chapter we have argued that the current philosophy towards automating office activities can, in systems supporting such activities, yield overall behaviour that can be hard to understand without the aid of some modeling tools. We have informally described the sort of systems that are of interest, shown what sorts of questions and problems can arise, and suggested an approach to answering these questions by studying message flow.

In chapter 2 we shall survey some of the recent literature to lend weight to our assumptions about how office activities will be automated. In chapter 3 we shall use these assumptions to develop a general formal model which we can use to discuss message flow and system behaviour without undue attention to system implementation (beyond our initial assumptions). In chapter 4 we shall then use this model in our efforts to arrive at useful methods for partitioning message domains into state spaces. The resulting message states will be the basis of our characterization of message flow and of global system behaviour.

In chapter 5 the information about message flow is further analyzed to answer questions about message loops, procedure loops and blocking. Chapter 6 consists of concluding remarks.

# 2. Background

In this chapter we survey the existing literature on office information systems. Through this effort we hope to establish what the term means, what characteristics such systems are expected to have, what assumptions are reasonable to make about their implementation, and what consequent analytic problems result from these assumptions.

In the first section we survey various discussions on the nature of office work and office systems. We try to come to a consistent view of the problems in applying computers to the field of office work, and we generate some principles concerning implementation. Office work is often "semi-structured", consisting of some simple, highly algorithmic parts, and some "fuzzy", judgemental parts. Exceptions and unsatisfied assumptions frequently arise. A successful computerized office system must be flexible enough to address these issues. Furthermore, office work changes frequently, or "evolves" which suggests that incremental development of automation is more appropriate than an abrupt replacement of an existing manual system by a computerized one. With some form of naive user programming, the automation of office procedures can be accomplished less painfully.

In the next section we survey the possible approaches to automation of office procedures. Several systems and programming languages are outlined and discussed in terms of the principles discovered in the previous section. Intelligent messages, automatic procedures, scripts and object-oriented programming languages each offer solutions to some of the problems to be solved. It is the approach of local, user-written automatic procedures that we choose to investigate further in this thesis.

## 2.1. What is Office Automation?

Broadly speaking, in order to be successful, an office information system must take an integrated approach to supporting office work. According to Hammer and Sirbu in their paper, "What is Office Automation?", many popular views of "office automation" are inadequate because they concentrate on too small an area, such as "the paperless office" or "office tools", and they fail to capture office work as a whole. They define "office automation" as: "the utilization of technology to improve the realization of office functions" [HaSi80]. That is to say, the task to be accomplished and the proposed solution must be viewed in the context of the entire office. All aspects of an office information system must be integrated in order to be able to exploit the interaction of related tasks and minimize the pain of augmenting or altering the system.

They define an "office procedure" as: "a structuring framework by means of which the individual tasks and activities performed by office workers are organized." The study of office procedures rather than just their constituent components highlights the *purpose* of the activities. These office procedures are typically semi-structured, that is, consisting of some routine, algorithmic parts and some "fuzzy" judgemental parts. They define an *office information system* as: "an integrated collection of components that supports the operation of an office procedure". Routine sub-tasks may be totally automated. Decision support tools may be provided for the "fuzzy" parts. In any case, an office information system is not a specific, all-

purpose cure-all, but it depends on the job to be done.

Ellis and Nutt in "Computer Science and Office Information Systems" define office information systems as: "entities which perform document storage, retrieval, manipulation and control within a distributed environment." [ElNu80] An automated office information system "attempts to perform the functions of the ordinary office by means of a computer system." They distinguish it from a data processing system in that it consists of "a collection of highly interactive autonomous tasks that execute in parallel". Hammer and Kunin state that, "An automated office system is an integrated and interconnected collection of components under the supervision of an intelligent control program" [HaKu80].

Morgan, in his paper, "Research and Practice in Office Automation" identifies most office functions as belonging to one of these categories:

1. Communications
2. Information acquisition, storage and retrieval
3. Data analysis and decision-making
4. Personal assistance
5. Task management

Communications involves the integration of electronic mail facilities, word processing, text formatting and message management. The second category can be seen as the extension of database management to textual, graphic and aural data. Decision-making can be aided by a rich collection of application programs for massaging and analyzing data. More intuitive user interfaces that provide us with models that correspond to the way we ordinarily think of real (as opposed to electronic) office objects can simplify the task of generating the information needed for the decision-making process. "Personal assistance" refers to a grab-bag of tools such as calendar programs and personal databases of phone numbers -- anything that helps you organize your time and your work. "Task management" refers to the automatic monitoring by the system of the tasks that are to be accomplished: since office work is typically event-driven, an office information system should be able to keep track of events and trigger new events or at least notify us when there is new work to be done [Morg80].

Office work is distributed in time and space and may involve the coordination of many parallel activities being performed by different people in different locations [HaSi80]. This work is "semi-structured" [SSKH82] in the sense that it is sometimes very regular and algorithmic. Although some of these activities may be routine to the point where a machine could perform them, much of the work is interactive and requires the attention of a human being [HaSi80]. Office work is riddled with unsatisfied assumptions. Human interaction is often necessary because an understanding of *goals* is required to accomplish a task. Therefore an office information system must be flexible, or "open-ended", support partial information, and handle missing information [FiHe80].

The office environment is constantly changing and evolving. Rather than instituting an office information system to suddenly replace the existing system, one should introduce it gradually by developing it incrementally [AtBS79, HaSi80]. This can be done by starting with something closely modeled on the existing manual system.

Ellis and Nutt point out that programming in an office information system is a special problem. Since these systems are developed incrementally, it must be possible to incorporate new code painlessly. The system must be capable of running procedures in parallel and in a distributed fashion. Furthermore, naive users must be able to have some direct control over the programs. This could be accomplished by generating code from user specifications, by using programming-by-example, or possibly by using syntax-directed program editors [ElNu80]. Lochovsky also points out that the approach of direct rather than procedural manipulation is especially appropriate for programming in office information systems [Loch83]. Fikes and Henderson suggest that other techniques arising from research in Artificial Intelligence may be well-suited to office information systems [FiHe80].

Because office information systems are susceptible to frequent change, modeling and analysis are likely to become very important in understanding and maintaining an evolving system [ElNu80]. Rulifson

predicts the advent of naive user-programming, and the growing importance of database and flow analysis, procedure analysis and system instrumentation. Systems will be built that analyze procedures and document flow and will be able to eliminate some procedures or reduce their complexity [Ruli79]. Hammer and Sirbu also state that an office information system will include "a variety of tracking and monitoring facilities that enable the procedure as a whole to be effectively managed and controlled." These facilities would be used, for example, for tracing document flow and evaluating the performance of a department. They also point out that "the best designed office procedures cannot account for all unusual situations that may arise ... thus, the office manager will have to possess both a global picture of the office procedures being performed, as well as a detailed understanding of the work of each of his individual subordinates." [HaSi80]

## 2.2. Automating office procedures

There are a variety of ways of automating office procedures. Ideally these techniques should be consistent with the principles established in the previous section. The alternatives differ mainly in who is able to specify automatic behaviour, and in who may be affected by this behaviour, directly or indirectly. Although different techniques are appropriate for solving different problems, we can evaluate their appropriateness to automating office procedures in terms of the criteria discussed above.

Let us consider, for example, the *ad hoc* approach to automating office procedures. This might be the case in a computerized office system where the automation of any given activity is implemented by a programmer. Since office workers cannot be expected to also be expert programmers, this means that a programmer must always be available to translate the specification of an office procedure into code. Office procedures, however, typically deal with documents (i.e. messages) that must pass through many different hands. These messages may have constraints on them that are not obvious to the implementor of the procedure. A programming error can have far-reaching side-effects. Since the implementor virtually has a free rein, we have almost no control over the changes made. Furthermore, since the changes are written in an arbitrary programming language, it can be arbitrarily difficult to evaluate them.

We have already established the need for an office information system to be an *integrated* system. Patched-in code could well undercut the system’s ability to keep all its parts functioning together properly. (A program that inadvertently bypasses a system log could cause a message to become "lost in limbo" to a tracing facility, for example.) The *ad hoc* approach fails on almost all counts: it is clumsy, error-prone, inflexible and difficult to evaluate. The only advantage is that there are no practical constraints on what you can do.

At the opposite extreme we have systems with all the automation built-in. The integrity of the system is ensured at the cost of flexibility. The problem here is that we must anticipate all future needs. Alternatively, changes to the system can be made only at a risk to system integrity, since reprogramming is required with the possible introduction of errors.

It follows that the only reasonable way to automate office procedures when automation is to be introduced incrementally and is expected to require fairly frequent alterations, is to have an extendible system. Some means are required for adding new procedures or altering old ones without jeopardizing the integrity of the system. If we are to have a flexible system capable of handling future needs without reprogramming, then there must be a means of translating specifications for office procedures into code that the system can understand. Who is responsible for the programming is another matter. This task may be associated with a system administrator or possibly with the users themselves. Which is the case depends on the nature of the application and the nature of the procedure to be automated -- clearly, if the task in question directly affects only a single user, then it would be highly desirable to have that person able to automate it himself.

We shall now look at a number of systems in order to establish what common features they have, and what characteristics we must capture if we are to model these systems.

### 2.2.1. SBA

The System for Business Automation (SBA) [deZl77] is an IBM research project built around the ideas of Query-By-Example (QBE). Query-By-Example is a relational database interface that allows users to submit non-procedural queries using example elements. A 2-dimensional skeleton of a table is displayed. One places example elements in the columns of the table, specifies conditions on these variables and indicates the ones to be displayed.

SBA extends this concept by allowing users to program with objects called "boxes" (related to *actors* [Hewi77]). A box has an identifier, input, output and contents. The input section consists of messages or boxes to be received and trigger conditions on the inputs. The output section describes the objects (*boxes*) produced and the messages sent. The specification of a box is done in a fashion similar to the specification of a QBE query.

### 2.2.2. OFS

A similar approach is taken in the Office Forms System (OFS) [TRGN82]. Users sign on to workstations that are connected by a network. They may create, file, retrieve, modify and mail electronic forms. Forms may also be "stapled" together into "dossiers" -- a device similar to a file-folder for keeping track of an arbitrary collection of forms. All forms have a unique system-wide key that can be used for tracing their passage through the system. OFS supports several different flavours of form fields. Fields may be modifiable at any time, or may be unmodifiable once they are filled, or may be required at the time of creation. Special fields are the date field, which is automatically written with the current date, and the signature field which is automatically filled with the user's name when some other field is filled or modified. In addition there are two flavours of automatic field which are computed as a function of any (physically) preceding fields on the form. One flavour will be evaluated only if all the argument fields contain some non-null value, and the other flavour accepts null fields.

Users may write automatic procedures that are triggered upon the receipt of mail. The trigger condition is specified by filling in a set of form templates (called *sketches*) with example elements (as in QBE). The actions for the procedure are specified by stepping through them manually with the form sketches. Since automatic procedures may create new form instances and mail them to other users, a chain of cooperating procedures can be created.

### 2.2.3. Officetalk

Officetalk [ElNu80] is another system based heavily on the idea of electronic forms. That is, it electronically captures those objects that are familiar to us in a real office. An Officetalk "desktop" has an "in-basket", an "out-basket", a "file index" (filing cabinet) and a "blank stock index". Users manipulate these objects directly with the use of a graphics mouse rather than through a command language. Officetalk supports the idea that one should *do* rather than *explain* what is to be done. The potential for forms and procedures using forms in office information systems has also been discussed in [LuYa81] and [Geha82].

### 2.2.4. Imail

Imail [HMGT83] is an intelligent mail system that places the intelligence with the messages. Messages are, in effect, simple programs that can convey information, engage in a dialogue with their recipients, and then take further action based on the responses received. Such messages can be used to gather information or automatically route themselves depending on the outcome of a dialogue. This approach to automation works well when the information to be managed is scattered in many different places. Automatic procedures, on the other hand, are more appropriate for managing and coordinating information that converges on a single location.

### 2.2.5. Smalltalk

A concept that is becoming increasingly popular as a programming tool for office information systems is the "object". There are many different working definitions of objects, but there are a few key ideas that they have in common. Objects are very similar to abstract data types. They have a static storage area or "memory", and a set of valid operations or rules for manipulating the object. This is also reminiscent of the construct of a *module*. Generally, however, one does not deal with objects by "calling" or "invoking" their rules but rather by sending them messages. In this way objects are all equal. They communicate with one another by passing messages rather than by calling and returning.

Smalltalk objects [BYTE81] are described by the *class* they belong to. An object's static memory is its set of *instance variables* and its rules are called *methods*. The methods describe the messages an object is prepared to accept. A method consists of (1) a pattern (describing the selector and its arguments), (2) temporary variables and (3) some *expressions* (i.e. actions). The expressions enable the object to (1) send messages, (2) assign variables, (3) return a value.

An expression consists of an object which is the *receiver* of the message, a *selector*, which specifies the kind of message sent, and optional *arguments*. If a value is returned by the expression, it may be assigned to a variable. Expressions may be composite. *Unary* selectors have highest precedence; next are *binary* selectors, and then *keyword* selectors, which may have several arguments. The expression "x + 2" sends the (binary) message "+ 2" to the object x. x presumably interprets this as an instruction to increment itself by the integer 2.

### 2.2.6. Oz

Oz objects [NiMT83] are similar to Smalltalk objects. They have *contents* and *behaviour* consisting respectively of a set of static local variables and a set of *rules* for accessing them. Rules have a set of associated conditions describing the circumstances under which they may be triggered. Those conditions may include the successful invocation of other rules in other objects. Rules may have the ability to spontaneously fire up when their trigger conditions become true without having to be explicitly invoked by another object. Oz is intended to be a prototype system for programming office information systems. Oz objects are sufficiently general to capture both office procedures (at one end of the spectrum) and "intelligent" messages (at the other end).

### 2.2.7. SCOOP

The System for Computerization of Office Processing (SCOOP) [Zism77] uses the *augmented Petri net* (APN) as the basic ingredient for specifying office procedures. This is a Petri net with each transition augmented by a set of production rules that indicate what actions are to take place when the transition fires. (See appendix C for a brief discussion of Petri nets.) An augmented Petri net may instantiate another augmented Petri net. This approach works well for fairly well-structured procedures that have a reasonably long life-expectancy. An augmented Petri net may "sleep" for hours or days or even months in a particular state while waiting for some event to occur. The construct is not intended for batch processing of high volume, highly-structured applications, nor is it appropriate for completely unstructured procedures (i.e. those handled by humans in an entirely *ad hoc* fashion). The Specifications Language for Office Procedures Execution (SLOPE) [Pott78] is a non-procedural interface to SCOOP.

### 2.2.8. Taxis

Taxis [MyBW80, Barr82] is a programming language for interactive database applications. Its fundamental objects are *classes* and *instances* of classes. Classes are instances of *metaclasses*. A class has a set of *properties*, much like a database relation has attributes. One class IS-A another class if it has "at least the properties" of the second class. This means that it may have additional properties, or it may have properties which are *specializations* of properties of the second class. *Student* is-a *person*, for example, with the additional properties of *student number*, *university* etc., and the specialized property *age* which is restricted to integers greater than 16, say. Taxis supports a variety of augmented Petri net called the Taxis *script*. Each transition in a Taxis script may have at most one set of conditions and actions, however, rather than a

list of condition/action pairs. Communication between scripts is allowed.

#### 2.2.9. BDL

The Business Definition Language (BDL) [HHKW77] is a very high level programming language for specifying data processing applications. There are three components to a BDL program. The form definition component defines the physical layout of documents, field names, their data types, and so on. The document flow component describes data flow with a directed graph with program steps and files as its nodes. A step is enabled if there is a document for each of its inputs (much like a Petri net transition). The document transformation component defines the actions of a program step on its input documents.

#### 2.2.10. OSL

The Office Specification Language (OSL) [HaKu80] is a formal language for describing office procedures. The two major components in a specification are the description of the application domain -- that is, the *objects* relevant to the application -- and a description of the procedures manipulating those objects. Objects are described using a variant of the semantic data model. The life-cycle of an object is managed by an initiating procedure, an administrative procedure that manages resources, and a terminating procedure that archives the object. The language has built-in constructs for describing how exceptions and special cases are to be handled.

#### 2.2.11. Information Control Nets

An Information Control Net (ICN) [Elli79, Cook80] is a formal model for capturing and analyzing information flow within offices. The model is useful for detecting deadlock, analyzing data synchronization and detecting communication bottlenecks. Some restructuring and streamlining of procedures can also be done within the model. A procedure can be displayed graphically as a collection of nodes representing *activities* connected by arcs representing precedence constraints. Activities may access a set of input and output repositories. This input and output relationship can also be represented graphically with arcs. Information flow and control flow are thus distinguished, being represented through two separate relations. Information control nets are capable of capturing parallelism, conflict and coordination as are Petri nets. Activities that can be eliminated or coalesced are detected by studying, for example, whether or not they permanently store any information, and whether precedence constraints are preserved after they are coalesced.

### 2.3. Summary

In this chapter we have surveyed the prevailing attitudes concerning automating office procedures. In particular, we have discovered that office procedures tend to be "semi-structured", consisting of some routine, algorithmic parts and some judgemental parts. Automated office procedures must be capable of integrating these parts. They must be capable of handling exceptions gracefully. The nature of the work often changes or evolves, and the system must be capable of incorporating these changes. Similarly, it is desirable if automated activities can be introduced incrementally. A useful way of accomplishing this is naive-user programming. Although this may be inappropriate for certain applications, there are many situations in which users should somehow be able to automate their own office procedures.

Typically, procedures appear to be event-driven. Events in these systems include the arrival of messages, the modification of forms and documents (i.e. user input), and the passing of time. In addition, procedures may need to coordinate several events by waiting, for example, for a collection of messages to arrive. Additional conditions may include constraints on the messages and on user input.

It is generally convenient to partition a system into workstations that have a function logically similar to desks. A workstation is associated with a single role, usually a single person. In many applications, office procedures may be associated with the workstations (this corresponds to the work associated with the role). One exception is the Imail system, in which automatic behaviour is associated with the messages themselves. When office procedures *are* local to a workstation, naive-user programming is a useful approach to automating them. The users, who are presumably the most knowledgeable about the task to be

automated, are thus able to directly specify the automation without the use of any intermediary (i.e. a programmer). These local, user-written procedures may consist of a single step that waits for a set of events (such as message arrivals) and takes some action on the objects collected when it is triggered. Alternatively, procedures may be scripts resembling Petri nets augmented with additional conditions and actions.

The literature on office information systems thus far has focused mostly on non-theoretical issues. Many papers have dealt with the requirements for a successful office automation system, and they have presented detailed arguments outlining the key areas of research. Other papers have provided practical solutions to some of these problems, notably: presenting convincing prototype environments for computerized office work; suggesting languages or frameworks for specifying and executing office procedures; and providing reasonable and intuitive interfaces for automated activities, such as programming-by-example in SBA. Finally, there has been some research into descriptive modeling of office information systems. These models are typically used to aid in the design of systems, or to provide a specifications language which may (possibly) be directly executed. With the exception of Information Control Nets, this author knows of no theoretical models for studying and analysing the behaviour of office information systems. This thesis addresses the latter problem by presenting a model for describing the behaviour of office systems with automatic message-handling, and by introducing some techniques for analysing instances of the model.

# 3. Message flow modeling

Before we can begin to address questions of global behaviour in message management systems, we need a formal framework for discussing automatic procedures. This framework must be powerful enough to capture quite general procedures but should be divorced from any particular implementation of them. It is immaterial, for example, whether procedures are written in some high-level programming language or in some intermediate code generated by a programming-by-example interface.

We will first present a model for describing messages and the procedures that manipulate them. Although we make some simplifying assumptions about procedures, we will show that quite general behaviour can be captured within the confines of our model. The terminology and notation introduced in this chapter is summarized for reference in appendices A and B.

The concepts of *message classes* and *message states* will then be introduced. Messages in the same class exhibit similar behaviour in that they potentially encounter the same sequences of procedures. Message domains are partitioned into state spaces in order to identify the message classes. Message paths traced by messages in the same class can then be thought of as alternating sequences of message states and procedures. The matter of how to partition the message domains is discussed in the following chapter.

## 3.1. Notation

We shall begin by introducing the notation for our model. Although some of the reasons for our choices of notation will be self-evident, others may not be immediately so. We shall discuss some of these choices in greater detail in the following section on modeling.

### 3.1.1. Locations

The logical configuration of an office information system is similar to that of a physical office. There are a number of workstations ("stations", for short), each of which is capable of communicating with any of the others. Whether or not the system runs as a collection of physically independent communicating machines or not is immaterial. Similarly the nature of the communication medium does not concern us here.

The collection of workstations is represented by:

$$S = \{s_1, \cdots s_N\}$$

In addition we have two *pseudo-stations*, $\alpha$ and $\omega$, that represent creation and destruction of objects. Creation and destruction are thus explicitly modeled. In some situations such stations will exist in truth: destruction of documents may in fact be implemented by permanently archiving them; also, creation of documents may be the responsibility of a privileged authorizing agent that assigns, say, unique identifiers. We require only that no messages be sent to $\alpha$ and that none be received from $\omega$. That is, they must behave as *source* and *sink*, respectively. The set of stations and pseudo-stations is:

$$S^+ = S \bigcup \{\alpha, \omega\}$$

Mailboxes are intermediate locations between stations. Messages passed between stations must be put into a mailbox just as physical documents are placed in an "in-tray". Although there may not be any "real" mailboxes in the system we are modeling, this allows us to distinguish between new mail and previously-seen messages. Furthermore, our model has one mailbox for every ordered pair of stations. This allows us to readily identify the sender of a message without having to resort to modeling a *sender* field for messages in transit. The latter approach would be entirely equivalent, however. The set of all mailboxes is thus:

$$M = \{m_{ij} | \ 1 \le i \le N, 1 \le j \le N\}$$

where $m_{ij}$ is the mailbox for messages sent from $s_i$ to $s_j$. Note that $\alpha$ and $\omega$ do not have mailboxes. A message "from" $\alpha$ appears at the station creating the message. A message that is destroyed goes directly to $\omega$. A station is allowed to mail messages to itself.

The set of all locations is

$$L = S \bigcup M$$

and, with the pseudo-stations:

$$L^+ = S \bigcup M \bigcup \{\alpha, \omega\}$$

The set of locations from which $s_i$ may receive messages is:

$$L(s_i) = \{\alpha, s_i\} \bigcup \{m_{ki} | \ 1 \le k \le N\}$$

This is the *local scope* of $s_i$ -- the locations that are accessible to the procedures at $s_i$. Messages may be created at $\alpha$, they may already reside locally at $s_i$, or they may arrive by mail from any of the $N$ stations (including $s_i$ itself, if desired).

Similarly $s_i$ may route messages to anything in the set:

$$R(s_i) = \{\omega, s_i\} \bigcup \{m_{ik} | \ 1 \le k \le N\}$$

(Note the reversal of subscripts on the mailboxes.)

### 3.1.2. Messages

Messages are assumed to be structured, and belong to one of several *message types* that encode this structure. The set of message types is:

$$X = \{X_1, \cdots X_K\}$$

The *domain* of a message type is assumed to be the Cartesian product of the attribute domains. (The attributes are the "fields" of a structured message.) We have, therefore:

$$dom(X_i) = \prod_{j=0}^{n_i} dom(X_{ij})$$

where $n_i$ is the number of attributes of message type $X_i$.

We reserve two attributes, $X_{i0}$ and $X_{i1}$ for the *identity* and the *location* of a message, respectively. The identity of a message instance is the only attribute that is never allowed to change. Since message instances may change value, we need some convention that allows us to keep track of their identity. We thereby also distinguish between a *message instance* and a *message value*: a message instance may assume different message values at different points in time. $dom(X_{i0})$ may be any enumerable set; for simplicity's sake we may assume it to be the set of positive integers. Of course, $dom(X_{i1}) = L$ (a message whose "location" is $\alpha$ or $\omega$ is not explicitly represented). A message value is represented by

$$x \in dom(X_i)$$

The $k$th attribute of $x$ is denoted by either $x_k$ or $x[k]$. (The latter notation is generally used when $x$ is the $j$th message in a tuple of messages, $\tau = (\ldots, x, \ldots)$, so $x = \tau[j]$, and $x_k = \tau[j][k]$. Message tuples are discussed below, in the section on procedures.) The identity of $x$ is $x_0$, and its location is $x_1$.

The *system state* is the collection of all the values of existing message instances. There is a set of message values $D_i$ for each message type $X_i$. The system state is:

$$D = < D_1, \cdots D_K >$$

where $D_i \subseteq dom(X_i)$. We do not represent messages whose "location" is $\alpha$ or $\omega$. Such messages have not yet entered, or they have already left, the system. We also insist that each $D_i$ contain at most one message with a given identifier, i.e.

$$\forall x \in D_i,\ y \in D_i,\ y_0 = x_0 \Rightarrow y = x$$

Identifiers are therefore unique within message types. By encoding type information into the identifiers, we could make them unique across all message types as well. Since we are generally interested in messages of specific types only, however, there is no real need to introduce this refinement.

In addition, we adopt the convention that

$$D(I) = D_i \text{ where } I = X_i$$

(i.e. if $I$ is an arbitrary message type then $D(I)$ represents the set of instances of that type).

### 3.1.3. Procedures

At each station $s_i \in S$ there may be a set of procedures that automatically process messages:

$$P(s_i) = \{ p_{ij} | 1 \le j \le k_i \}$$

where $k_i$ is the number of procedures at $s_i$. The set of all procedures is:

$$\begin{aligned} P &= \{ p_{ij} | 1 \le i \le N, 1 \le j \le k_i \} \\ &= \bigcup_{i=1}^{N} P(s_i) \end{aligned}$$

Every $p \in P$ has a set of *input types*, *trigger conditions* and *actions*. A procedure (within our model) is a single-step activity. A collection of messages (inputs) matches the trigger condition and the actions are performed, causing messages to be modified (possibly created or destroyed) and routed. The input types are the types of the messages $p$ needs in order to evaluate its trigger conditions:

$$I(p) = < I_{p1}, \cdots I_{pl_p} >$$

where $I_{pi} \in X$. $l_p$ is the number of inputs to $p$.

The inputs to a procedure $p$ form a set, or rather a tuple, of messages that we call an *input tuple*. We usually represent such a tuple by the symbol $\tau$, where $x = \tau[j]$ is the $j$th input message and $x_k = \tau[j][k]$ is the $k$th attribute value of the $j$th message. Such a tuple $\tau$ may trigger procedure $p \in P(s_i)$ if $\tau \in \prod_{j=1}^{l_p} dom(I_{pj})$ and it satisfies the trigger conditions of $p$. In addition, the messages in $\tau$ must be available to $p$, that is, $\tau[j][1] \in L(s_i)$, and each of the messages in $\tau$ must be unique (a message can't play two roles for a single procedure). We formalize this in the set $T(p)$ of message instances that may trigger $p \in s_i$, where:

1. $T(p) \subseteq \prod_{j=1}^{l_p} dom(I_{pj})$

2. $(\tau \in T(p)) \wedge (I_{pj} = I_{pk}) \wedge (\tau[j][0] = \tau[k][0]) \Rightarrow j = k$
3. $\tau \in T(p) \Rightarrow \forall j\ \tau[j][1] \in L(s_i)$

Tuple $\tau$ can thus *trigger* $p$ if $\tau \in T(p)$ and for all $I_{pj} \in I(p)$ we have $\tau[j] \in D(I_{pj})$ or the $j$th message is to be created by $p$ (i.e. $\tau[j]$ does not exist yet). We then say that $p$ is *enabled*.

In order to disambiguate conflicts between procedures, we allow for a partial ordering "$\gg$" of procedures. If both $p$ and $p'$ are enabled and $p \gg p'$, then procedure $p$ must be fired. We say that *$p$ has priority over $p'$*. $p'$ may only be fired if it is enabled and $p$ is not. This is useful if $p$ is triggered when message $x$ matches some coordinating message $y$ and $p'$ is triggered when there is no coordinating $y$. Without partial ordering of procedures it would be impossible to express the condition: "fire $p'$ with message $x$ only if there is no matching message $y$". For example, if procedure $p$ matches inventory forms to order forms and $p'$ looks for order forms for non-existent items, then the only way to capture the trigger condition of $p'$ is to have it accept all order forms not accepted by $p$.

Actions map input tuples to output tuples. In our model, there is a one-to-one correspondence between input messages and output messages *even if the procedure creates or destroys some messages*. This is why we need the pseudo-stations $\alpha$ and $\omega$. They allow us to (somewhat artificially) model messages that are created as arriving from $\alpha$, and those that are destroyed as being sent to $\omega$.

The action of procedure $p$ is a mapping:

$$A(p): T(p) \rightarrow \prod_{j=1}^{l_p} dom(I_{pj})$$

such that the identities of input messages are never changed, and they are routed only to valid locations. We use the notation $a_{jk}$ to refer to the individual attribute mappings of $A(p)$. If $\tau' = A(p)(\tau)$, then

$$a_{jk}: \tau \mapsto \tau'[j][k]$$

For each $j$, therefore, $a_{j0}$ is the identity map (can't alter identity of $\tau[j]$). Also, the $a_{j1}$ are the *routing functions*, since they are responsible for updating the location attributes. Clearly, the domain of $a_{j1}$ is $R(s_i)$, where $p \in P(s_i)$.

Within our model, user input, external databases and other outside sources of information are not explicitly represented. When procedures make use of external information, we consider the mappings of the procedures to map to a *choice* of possible values (modulo the outside information sources). The $A(p)$ are therefore not necessarily well-defined (i.e. functions). Consequently, when we perform our analysis with traditional machine models such as finite automata and Petri nets, a certain amount of non-determinism appears that may not necessarily be evident in the system under analysis. A function that sets a field of a message to anything a user wishes to enter is therefore modelled as a mapping from the input message to the entire domain of that message field. We should therefore keep in mind that this "non-determinism" is often an artifact of our attempt to exclude arbitrary information sources from the outside world.

If $\tau$ triggers $p$ then the system state $D$ is updated to reflect the firing of $p$. Input message instances are replaced by their new values. If $\tau' = A(p)(\tau)$, then the new system state $D' = < D'_1, \cdots D'_K >$ is defined by:

$$D'_i = (D_i - \{\tau[j] | I_{pj} = X_i\}) \bigcup \{\tau'[j] | (I_{pj} = X_i) \wedge (\tau'[j][1] \neq \omega)\}$$

Messages that are destroyed are simply deleted from $D'_i$.

### 3.2. Modeling power

The purpose of our model is to provide a precise context for our discussions of message flow and of global behaviour. First, we must convince ourselves of the appropriateness of the model. To do so we may ask, "What kinds of systems are we able to capture?", "What are the inherent assumptions and limitations of the model?", "What properties does the model exhibit?" and "How do systems correspond to the model?"

The surface assumptions that we have made are that there are discrete "locations" that have control over messages, that messages are structured but have little or no built-in intelligence, and that procedures are stationary, they reside permanently at the locations, and they are effectively "memoryless" (though we shall see that this is not really a restriction).

### 3.2.1. Locations

Locations in our model are either workstations or mailboxes. Workstations serve to subdivide a system so that every message and every procedure "belongs" to exactly one station. This corresponds strongly with the idea of physical documents having a unique location and a unique person in control of it at any time. We have provided distinct mailboxes for each ordered pair of stations in order to more easily identify the source of messages. Without these distinctions we would need to explicitly include a *sender* field with all messages. Naturally there need not even be mailboxes in the real system. In that case, the mailboxes of the model would simply exist to represent the fact that a message has been sent but not yet seen by its recipient. Conceivably one might wish to further subdivide stations into "directories" for the purpose of organizing groups of messages. Such an extension could easily be made to the model by allowing for a set of "directory locations" belonging to any given station. For simplicity's sake, however, we shall presently restrict the locations to workstations and mailboxes.

### 3.2.2. Messages

Messages are "structured" in our model. A collection of messages of the same type is equivalent to a database relation where each message is a tuple in the relation. The domain of a message type is assumed to be the Cartesian product of its attribute domains, but we do not make any assumptions about the attribute domains other than that values in the domain be finitely representable in a computer. Messages could therefore be composed of text fields, graphical images, and so on. Messages are assumed to contain a finite number of attributes, but most documents (such as forms) can reasonably be expected to satisfy this restriction. Note, however, that an attribute could be a text field allowing, for example, a thirty-page report to be a "field" of a message.

### 3.2.3. Procedures and multi-step activities

Procedures are assumed to be "personal" -- that is, they reside at a single workstation and do not look at messages at other workstations. Other kinds of automated activities may, of course exist, such as the "intelligent messages" described previously. We are interested, however, in procedures that handle volumes of messages passing through individual stations, rather than procedures that move from station to station. This corresponds more closely with our idea of workers having a fairly well-defined domain of responsibility.

The procedures of our model are not necessarily intended to completely represent an automated activity. A procedure may be just a single step within a more complicated task to be accomplished. Furthermore, these procedures do not have any "memory", though it is clearly useful to be able to "remember" a message that has been seen before. There is a difference, though, between automated activities and our usual way of thinking of computer programs. When a program is executed, it is allocated some memory, given some processing time, and it runs to completion. There may be some side effects, but generally the traces of the program's execution disappear. An automated activity may, however, be interrupted and have to wait for a long time (minutes, days, months ...) before it can resume. The "state" of the activity must be remembered if it is to continue from where it was before. This information strikingly resembles a message, since there is typically a well-defined list of things to remember. The state of the activity can thus be passed from procedure step to procedure step as a message, without the need for any other special mechanism for remembering information. Furthermore, an activity that *does* span several workstations can be captured by modeling all information exchanged as messages. There is thus no need for procedures to have a special "memory".

To demonstrate how useful this idea is, consider an activity that is structured as an augmented Petri net. The "state" of a Petri net is its *marking*, the number of tokens in each of its places (see appendix C).

(The places of the augmented Petri net may or may not be interpreted; in the Petri net of procedure interactions described in chapter 1, the places represented the presence of a message, but within the automated activity they may well have a very different interpretation or possibly none at all.) If the transitions of the activity represent indivisible "steps", then we may translate them to procedures in our model. A new message type is added to the system representing the "state" of the augmented Petri net (i.e. of the automated activity). The state of the augmented Petri net consists of the current marking of the underlying Petri net, the identities of the messages it modifies and the values of any static variables that are to be "remembered". The steps then translate into procedures with inputs as before, one per original message type, plus an additional input for the message that stores the state of the activity. The trigger conditions of the procedures in our model would encode the markings for which the transitions of the augmented Petri net would be triggered.

As an extremely simple example, consider the diagram in figure 3.1. This activity is used to track down documents. Transition $t_1$ allows one to generate a request for information about a document. One must supply a short description of the request and someone to whom the request should initially be sent. Place $p_1$ represents a request having been created. Transition $t_2$ causes the request to be mailed. Place $p_2$ represents an outstanding request for which a response is awaited. A response may be affirmative ("yes -- the document is ..."), negative ("no -- I have no idea") or tentative ("I dunno, but why don't you ask ..."). If the response is tentative, $t_3$ fires, sending a new request to the named user. If the response is negative or affirmative, then $t_5$ fires and reports the response to the initiator of the request. If no response is received within some reasonable time limit, then $t_4$ fires and issues a reminder.

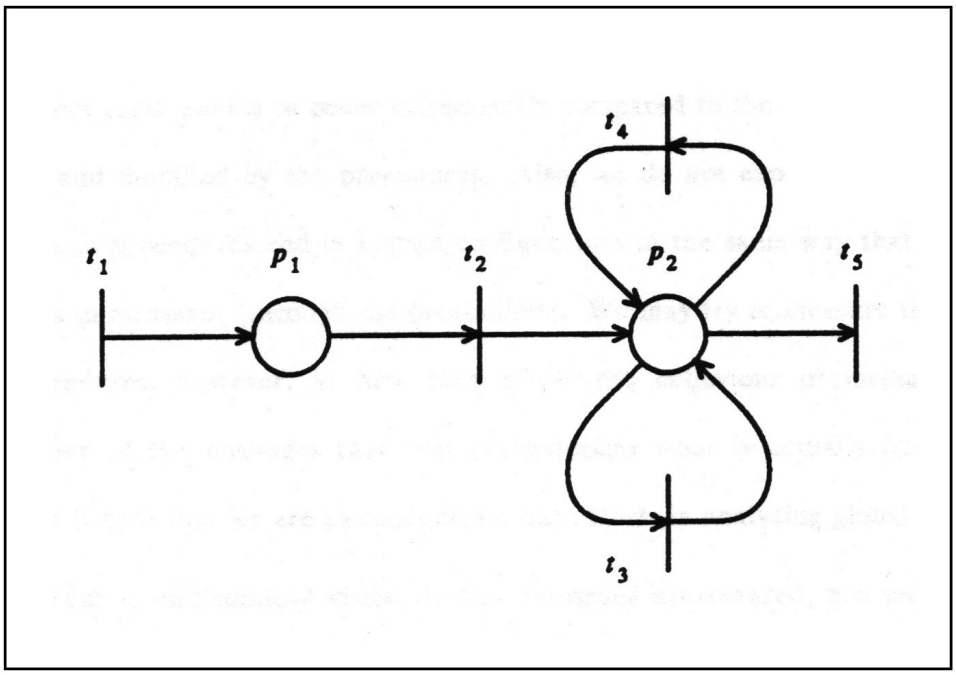



**Figure 3.1** : An office activity

The state of this activity includes the marking of the Petri net, the identity of the request/response message being awaited, and the static variables containing the text of the request, the identity of the person to whom the request was issued and the time that the request was mailed. Within our model the transitions of the Petri net are represented by procedures that are triggered by the presence of the request/response message and another message that encodes the state of the activity. The single exception is $t_1$ since it is the step required to initiate the activity -- before it is fired there *is* no activity state message.

By thus recording the state of a multi-step office activity in the contents of a message used exclusively by the steps of the activity, and by translating those steps into procedures triggered by the presence of the input messages and the activity state message we can capture the notion of activities with "memory". Furthermore, it is even possible to distribute such activities across several workstations by mailing the state of the activity like an ordinary message.

## 3.3. Message paths

Our model of message management views procedures and locations as being basically static entities. Although procedures are altered and workstations may be added to a system, we expect these events to occur infrequently compared to the rate at which messages are processed and modified by the procedures. Also, we do not expect to be able to formalize the changes in procedures and in system configuration in the same way that we can formalize the changes in messages (through the procedures). We may try to measure the large-scale changes in procedures, however, in how they effect the behaviour of messages. Since it is the behaviour of the messages that best characterizes what is actually happening on a *regular* basis, it is here that we are to concentrate our efforts in analyzing global behaviour.

What is immediately visible is that messages are created, are modified and routed by sequences of procedures at different workstations, and are eventually destroyed. We can think of messages as tracing a *path* through the network of stations as they encounter different procedures. In between the procedures they acquire different *values* (including their *location*) which they hold until the next procedure changes their value. We may thus think of a *message path* as being not merely a sequence of procedures encountered by the messages, but as an alternating sequence of values and procedures. This message path is an expression of "message flow" since it encapsulates all the locations a message visits during its lifetime, especially if we allow ourselves to think of procedures as extremely brief, temporary "locations".

A certain amount of artificial non-determinism appears in the message paths due to coordination. A message may be handled by any number of procedures depending on the coordinating messages that it encounters at those procedures. An order form may be routed in different directions, for example, depending entirely on whether there are sufficient items in stock listed on the corresponding inventory record. We cannot, in general, predict in advance what values those coordinating messages will have, so we cannot predict what path a message will trace. It is conceivable, however, that we may be able to determine what *set* of message paths a given message may trace. To do this one would have to list the procedures that might handle a message, then, for each procedure, list all possible values that it could acquire next, and so on. Depending on whether or not there are unique coordinating messages at run time, the action performed may or may not be uniquely determined. In either case, however, when we restrict our model to a consideration of a single message type at a time (as we do with message paths), the actions performed are not deterministic.

### 3.3.1. Path-equivalence

To completely characterize the behaviour of messages of a single type, one would have to do this for all possible initial values of such messages. We may be able to simplify this task by considering equivalence classes of messages. Two messages are deemed to be *path-equivalent* if the sets of possible procedure sequences they may encounter are equal. That is to say message $x$ is *path-equivalent* to message $y$ if $x$ can potentially encounter any sequence of procedures that $y$ can potentially encounter (given the right coordinating messages), and vice versa. We may formalize this as follows:

Suppose $x \in dom(X_i)$. Let

$$\hat{p}(x) = \{ p \in P | X_i \in I(p),\ X_i = I_{pj},\ \exists\, \tau \in T(p) \text{ such that } x = \tau[j] \}$$

i.e. $\hat{p}(x)$ is the set of procedures that may be triggered by $x$.

$$\hat{a}_p(x) = \{ A(p)(\tau)[j] | p \in \hat{p}(x),\ \tau \in T(p),\ X_i = I_{pj},\ x = \tau[j] \}$$

i.e. $\hat{a}_p(x)$ is the set of values to which $x$ may be mapped after triggering $p$.

Finally, we wish to define $\hat{l}(x)$ as the set of possible sequences of procedures that $x$ may encounter. That is, $\hat{l}(x)$ is the language (i.e. set of strings) over the alphabet $P$ of procedures representing the possible sequences of procedures that $x$ may trigger. We may define $\hat{l}(x)$ recursively as:

$$\hat{l}(x) = \begin{cases} \{p\hat{l}(x') | p \in \hat{p}(x),\ x' \in \hat{a}_p(x)\} & \text{if } x_1 \neq \omega \text{ and } \hat{p}(x) \neq \varnothing \\ \lambda \text{ (the empty string)} & \text{otherwise} \end{cases}$$

We call $\hat{l}(x)$ a *message flow language*. Sequences terminate, of course, when messages are destroyed, since no procedure can be triggered by a message whose location is $\omega$. Some strings in the languages may be infinite, if messages never get destroyed.

Now we may say that

$$x \tilde{} y \text{ if } \hat{l}(x) = \hat{l}(y)$$

that is, $x$ and $y$ are *path-equivalent* if they potentially trigger the same sequences of procedures.

Our hope is that we may be able to simplify our problem of capturing behaviour in terms of message flow by limiting the number of messages that we must examine. If we can separate messages into equivalence classes, we may be able to save some work and obtain a cleaner description of behaviour by obtaining expressions of message flow for the classes. Unfortunately this is not an easy problem. We can show that it is possible for a message system to simulate two Petri nets at once so that two messages are path-equivalent if and only if the corresponding Petri net languages are equivalent. Since the latter problem is undecidable, so is determining path-equivalence. The proof follows.

### 3.3.2. Undecidability of path-equivalence

The following definition is from [Pete83]:

**Definition 3.1** : A language $L$ is an *L-type Petri net language* if there exists a Petri net structure $(P, T, I, O)$ (as defined in appendix C), a labeling of the transitions $\sigma: T \rightarrow \Sigma$, an initial marking $\mu$, and a finite set of final markings $F$ such that $L = \{\sigma(\beta) \in \Sigma^* | \beta \in T^* \text{ and } \delta(\mu, \beta) \in F\}$. (The next-state function, $\delta(\mu, t) = \mu'$, is extended to sequences of transitions in the obvious way, i.e. $\delta(\mu, t\beta) = \delta(\delta(\mu, t), \beta)$.)

The language $L$ is therefore obtained by mapping transition firing sequences to strings over $\Sigma$ via the labeling function. Of course, the transition firing sequences themselves form a Petri net language over $T$ by using the identity map as the labeling function. Note that through the permissive nature of Petri nets, one is not obliged to stop firing transitions when a "final" marking is reached. One may continue firing transitions, passing through final markings as often as possible. The definition of the language states only that we must be in a final marking when we do decide to stop.

There are 12 classes of Petri net languages described in [Pete83] each with slightly different definitions. We are interested here in the L-type languages that allow non-distinct but non-null labeling of transitions. (These are known as the non-$\lambda$, non-free labelings.)

**Lemma 3.2** : Every L-type Petri net language is a message flow language.

**Proof** : By construction. (When symbols in the Petri net notation are identical to those in our notation, we shall distinguish them with an accent, for example $P$ for Petri net places and $P'$ for procedures.) Let $L$ be a Petri net language as in definition 3.1. Let the set of procedures $P' = \Sigma$. Consider message types $X_1$ and $X_2$ as follows:

Let message type $X_1$ have one attribute for each place in $P$ in addition to its identity and location. The domain $dom(X_{1j})$ of each of these attributes is the set of non-negative integers. A marking $\mu$ is thus represented by a message $x$ with attribute $\mu_j = x_{j+1}$ ($x_1$ being the location of the message).

In addition, let message type $X_2$ have an attribute $X_{22}$ whose domain is $T$, the set of Petri net transitions, and an attribute $X_{23}$ whose domain is { "yes", "no" }. (This attribute will be used as a "toggle" to decide whether to stop or to continue when we reach a "final" marking.)

Let $I'(p) =< X_1, X_2 >$ be the set of inputs types for each $p$. Let $T'(p) \subseteq dom(X_1) \times dom(X_2)$ be defined by:

$$T'(p) = \{(x, y) |\ x \text{ represents } \mu,\ y_2 = t,\ \sigma(t) = p,\ t \text{ is enabled in } \mu\}$$

That is, $p$ is triggered if $p$ is the label for some transition $t$, and $t$ is enabled in the marking $\mu$ that $x$ represents. The message $y$ exists merely to *select* which transition is to be used, since $p$ may be the label for several.

The action of $p$ modifies $x$ to represent $\mu'$ where $\mu'$ is the marking that results from firing transition $t$ in marking $\mu$. In addition, if the marking $\mu' \in F$, then $p$ may conditionally destroy $x$ if $y_3 =$"yes". This is necessary since the definition of a Petri net language allows us to optionally continue firing transitions even after reaching a "final" marking.

It should now be clear from the construction that $\delta(\mu, \beta) \in F$ if and only if $\sigma(\beta)$ is a potential procedure firing sequence that destroys message $x$ representing $\mu$. We need only supply one message $y \in dom(X_2)$ for every combination of $t \in T$ and "yes" or "no". The non-determinism of our message model guarantees a choice of procedure firings for every corresponding choice of transition firings. Consequently $L = \hat{l}(x)$. □

Note that message type $X_2$ is not strictly required, since we may construct a single message type that combines the attributes of both $X_1$ and $X_2$. A new "toggle" attribute would be used to tell us which of the two old types are represented by an instance of the new type. The result can thus hold even in systems where there is only a single message type.

We may now state the main result:

**Theorem 3.3** : Path-equivalence is undecidable.

**Proof** : Let $L$ and $L'$ be two Petri net languages. Construct a message system that simulates *both* Petri nets at once by letting $X_1$ have enough attributes for the places of both nets and a "switch" attribute, $X_{switch}$ that tells the procedures which net to simulate. The switch attribute is set at the time of message creation and must not be changed. A message can thus represent a marking in either net depending on the switch setting. (Obviously, messages are not allowed to hop from one Petri net to another, so the switch may not be changed.)

Procedures are triggered by messages that represent markings in which transitions for the appropriate net are enabled (as above). The trigger conditions for the procedures are of the form:

$$\begin{aligned} T'(p) = \{(x, y) | x \text{ represents } \mu,\ x_{switch} = L,\ y_2 = t,\ \sigma(t) = p,\ t \text{ is enabled in } \mu\} \\ \bigcup \{(x, y) | x \text{ represents } \mu',\ x_{switch} = L',\ y_2 = t',\ \sigma(t') = p,\ t' \text{ is enabled in } \mu'\} \end{aligned}$$

Actions similarly map $x$ depending on the setting of $x_{switch}$.

Now consider message $x$ such that $\hat{l}(x) = L$ and message $x'$ such that $\hat{l}(x') = L'$. Such $x$ and $x'$ exist by the construction of *Lemma 3.2*. Clearly $x$ is path-equivalent to $x'$ if and only if $L = L'$. Since the latter problem is undecidable [Pete83], so is the former. □

The P-type Petri net languages include strings $\sigma(\beta)$ obtained from all $\beta$ such that $\delta(\mu, \beta)$ is defined, that is, from *all* transition firing sequences, not just those that end in some "final marking". It is also undecidable whether P-type languages are equivalent. If we extend message flow languages to include procedure firing sequences that do not result in the destruction of a message, then path-equivalence is consequently still undecidable.

### 3.3.3. Message states

Since path-equivalence is undecidable, it appears that we are asking too much if we expect to obtain an exhaustive characterization of message flow. Nevertheless we may be able to make use of a weaker form of path-equivalence. One of the reasons that we run into problems is that Petri nets can have an infinite number of states. We are required to keep track of too much information about the value of a message if we are to characterize the paths it may take. Since there is already a large degree of uncertainty built into the message paths (due to coordination), it hardly seems worthwhile to distinguish so sharply between

different message values. Furthermore, since we cannot effectively analyze message paths when we do keep such detailed information, we are forced to make some simplifications.

Naturally one simplification is to limit the power of procedures. In chapter 4 we shall consider procedures of varying degrees of complexity. We would still like to be able to obtain some results for more general procedures, so the approach we take is to try and limit the number of "message states" that we need to consider. We may do this by partitioning message domains into a finite number of *message states*, each of which represents a collection of "similar" messages. We thus eliminate the need to consider a potentially infinite number of message values, and we obtain message paths that are considerably less complex. Procedures will map messages from one state to another if there is any message value in the first state that it maps to some value in the second state. Clearly the art is in choosing the partition of a message domain in such a way as to lose as little information as possible.

An obvious first attempt at such a partition is to allow one message state for each station. All messages at a given station (or any of its mailboxes) would then be considered equivalent. This naturally corresponds to the notion that procedures at a station can only be triggered by messages owned by that station. It is easy to see, however, that this would be inadequate to handle the case where different procedures at the same station have different ways of handling similar messages from different sources (such as in our example from chapter 1 in which new order forms to be filled come from one source and those that have already been filled come from another source).

Our next attempt might be to have a message state for each location, thus distinguishing between messages from different sources. Messages arriving from the same source may, however, be interpreted differently depending on their value. We would therefore like to include as much information as we "reasonably" can, yet still have a finite ("reasonably small") number of message states to consider. In the next chapter we shall consider ways of producing a reasonable partition.

### 3.4. Summary

The message system model presented in this chapter allows us to describe the complex interactions of office procedures. We represent messages, stations, mailboxes and procedures. Messages are assumed to be structured and resemble tuples in a relational database. Messages of the same type have the same structure. Message instances also have a unique identity and location, the former being unalterable and the latter being either a station or a mailbox.

Procedures are associated with stations, and may manipulate messages at that station or one of its mailboxes. Procedures have a fixed number of inputs of given message types. A procedure may be executed if its trigger condition holds over some set of input messages of the required types. The action of the procedure enables it to alter, create, destroy and route any of its inputs. One may place priorities on procedures to disambiguate situations where two or more procedures may be triggered by the same input set.

The most severe limitation of this model appears to be that procedures have no memory. In fact, however, it is possible to "simulate" the memory of a procedure by adding a new message type. Complex, multi-step activities can be modeled by a collection of single-step procedures (possibly residing at more than one station) and a single "message" that is used to record the state of the activity. We have shown how one popular scheme for describing office activities, the augmented Petri net, can be captured within our model.

With our model we can easily display such properties as parallelism, non-determinism, synchronization (coordination of messages) and conflict (competition of procedures for messages).

There is no explicit mechanism within the model for adding stations or altering procedures. Rather, the model is intended to represent system behaviour for a given configuration of workstations and procedures. Changes in the configuration are interpreted as changes in the model. We cannot, of course, predict what changes in configuration are to be made, but we can predict the behaviour of the system for a given configuration. We attempt to characterize this behaviour in terms of the objects that are modified in a regular way, namely, the messages.

Our intuitive notion of message behaviour is the flow exhibited as messages are shunted from procedure to procedure and as their values are changed in the process. We try to characterize message flow by the path that it traces: an alternating sequence of values and procedures. We would like to classify messages according to these paths by grouping together those that have the potential for encountering the same sequences of procedures. We show, however, that this notion of equivalence is too strong, and that we therefore cannot hope to characterize message behaviour in this way.

We may nevertheless obtain some results by simplifying our problem somewhat. If we can partition message domains into finite state spaces then we may be able to obtain paths in terms of "message states". Although we lose some information by grouping messages that may not be capable of following precisely the same paths, we hope to gain by achieving an expression of message flow that provides us with some insight into the behaviour of a message system as defined by the procedures in it. In the following chapter we shall investigate the problem of finding a "good" way to partition message domains.

# 4. Message flow

In the preceding chapter we developed a notation for describing automatic behaviour in message systems, and we discussed the idea of message flow as a measure of system behaviour. Since it is not, in general, possible to exhaustively enumerate all potential message paths, we must seek some less demanding way of describing message flow. The difficulty in classifying message paths seems to be a consequence of the number of values that messages of a given type may acquire being practically unlimited. We are reluctant, however, to consider only messages with finite domains, or to consider only severely restricted procedures. Our approach to this problem is to consider only a finite number of *sets* of message values by partitioning message domains into finite state spaces. A given message value would therefore be in only one of a finite number of *message states*. This allows us to consider large or infinite domains as being effectively finite. As we shall see, this finiteness makes our analysis of message flow tractable by sacrificing some information.

In this chapter we will consider ways of developing a reasonable partition of message domains by classifying different kinds of trigger conditions and actions. "Control attributes" appearing in trigger conditions and actions are used to partition attribute domains and message domains. Algorithms and techniques for collecting the necessary information about control attributes are presented. The message domain partitions may then be used to develop a finite state machine model of message flow. "Symbolic messages" are introduced as a technique for gathering the raw message flow data. These data may then be translated into message flow expressions that describe the state transitions that may take place.

### 4.1. Message paths and states

In this section we shall consider the effect that trigger conditions and actions have upon message flow. By studying and classifying triggers and actions, we hope to gain some insight into the matter of defining message states. We will develop the notions of "selection attributes", "routing attributes" and "control attributes" as being crucial to an understanding of message flow. These attributes appear in the trigger conditions and actions, and either directly or indirectly influence the path that a message takes. Later in this chapter, after we consider the different ways these attributes affect message behaviour, we will show how to use these ideas to generate finite state models of message flow.

In chapter 3 we defined a "message path" as an alternating sequence of message values and procedures. The potential sequences of procedures alone became what we called the "message flow language" of a particular message instance. Messages that could potentially encounter the same sequences of procedures were said to be equivalent with respect to message flow.

Since we have shown this notion to be too strong to be of any practical use, we shall attempt to weaken it slightly in order to get some usable expression of message behaviour. By partitioning message domains into a finite state space we limit the possible combinations of messages and procedures to be considered. Furthermore, since procedures can be thought of as effecting transitions of messages from state to state, we can derive a finite state machine representation of message flow. We can thus extend the notion of

message paths to be alternating sequences of message *states* and procedures. As finite state machines are a well-understood formalism, this leads to a classical interpretation of system behaviour.

There are, however, two rather obvious problems. The first is a consequence of the fact that we cannot exhaustively enumerate all message paths. This means that the paths that we derive in terms of message states must of necessity sacrifice some information. We must therefore be cautious in our interpretation of the derived message paths. The second problem is how to choose the state spaces so as to minimize this loss of information.

Clearly, triggering conditions are important in deciding what procedures a message may trigger. If we partition message domains according to trigger conditions alone, then messages in the same message state are capable of triggering the same procedures. Two messages that do trigger the same procedure may be routed to different stations, however, so the procedures that they trigger next may well be different. We would like to partition message domains finely enough that messages in the same block of the partition potentially encounter the same procedures indefinitely, or until they are terminated. (This is, as we stated already, impossible, but we shall try to obtain a reasonable approximation.)

The set of procedures that a message may trigger is directly affected by its value and its location. The value of the message affects triggering by satisfying or not satisfying trigger conditions. The location may be examined by a trigger, but it is also important in limiting the procedures that may access the message at all, independent of the trigger conditions. (Even if none of the trigger conditions of any procedure discriminate between messages on the basis of location, the location of messages they handle still limits when procedures may process them.) Collections of messages of the same type are thus "split" along different paths by:

1. matching different $T(p)$
2. being routed to different stations by $A(p)$

In turn, the actions of the procedures that handle a message determine its subsequent value. Actions, unfortunately, create problems for us. Suppose that actions were not allowed to alter messages, but only to route them (that is, they may only change their *location* attribute). If there are a finite number of procedures, then for a given message value we can decide what procedures it may trigger if it arrives at the location belonging to the station. Since the value of the message may not change, we know for all time what procedures it may trigger. There being only a finite number of locations, we know that any message may reach only a finite number of states. Its behaviour with respect to message flow is thus comparable to that of a finite state machine. It is simply a matter, then, of enumerating all possible combinations of triggering conditions and tracing the procedures that messages matching them will encounter. We may start by assuming no knowledge about a message, progressively adding constraints on the message as we consider its encounter with procedures and other messages.

If, on the other hand, actions are allowed to arbitrarily alter the value of the message, then the information that we gather about the message (i.e. that it satisfies certain trigger conditions) will be (partly) destroyed. We must therefore try to glean as much as possible from the actions by using the information gathered to determine what new values the message may have.

Since message states are obtained by partitioning message domains, any given message state $\sigma$ for messages of type $X_i$ must satisfy:

$$\sigma \subseteq dom(X_i)$$

Specifically, we are interested in message states that are obtained by partitioning the attribute domains:

$$\sigma = \prod_{j=0}^{n_i} R_j$$

where each $R_j \subseteq dom(X_{ij})$ is a block in some partition of $dom(X_{ij})$. Since messages are modified attribute by attribute, this kind of message state makes it simpler to keep track of the current state of a message.

Since we plan to derive message states partly from trigger conditions, the $R_j$ will often correspond to simple conditions on the $j$th attribute. If $X_{ij}$ is a numeric field, the conditions that express $R_j$ may specify a *range* of values in $dom(X_{ij})$ (hence our use of the symbol $R_j$).

One attribute domain that is trivial to partition is the location of messages. Recall that procedures at a given station may only access messages that "belong" to that station by either residing at the station or at one of its mailboxes. One may therefore partition locations into groups, one per station. Furthermore, since there are only a finite number of locations, we may even go so far as to distinguish between all locations. We shall henceforth assume, therefore, that all messages in the same state are at the same location, i.e. for any message state $\sigma$, we have:

$$\text{for all } x, y \in \sigma \Rightarrow x_1 = y_1$$

Before we take a closer look at the nature of triggers and actions, we shall introduce some definitions that will help us in deciding exactly what attributes are of interest to us.

### 4.1.1. Control attributes

We need not necessarily consider all message attributes when we partition our message domains into a state space. Some attributes may not affect the path of messages at all. Attributes that do affect the path do so by affecting either the triggering of procedures or the routing of the message.

To begin with, although the domain of a procedure's actions and triggers is all of $T(p)$, it is in fact likely that only some of the attributes of the input messages are examined or modified. We would like to identify the *true* arguments of a function as the ones that are actually used in the computation of the value returned. We are assuming, of course, that all the functions we will be dealing with are effectively computable, and describable by algorithms. A procedure that increments a field of a message clearly does not need any of the information contained in the other fields of the message in order to compute the result. The only true argument to the incrementing function is therefore the field that is modified.

The true arguments to a function can generally be determined by inspection. For example, the true arguments to $f(x, y, z) = x^2 + y$ are clearly $x$ and $y$, provided the domains of $x$ and $y$ have more than one element. (We note that it is possible to construct odd functions for which the "true" arguments are debatable, such as $f(x) = x/x$ where $x$ is a positive real. Although $x$ appears to be a true argument (it occurs in the definition of the function), in fact it is not since $f(x) = 1$ is an equivalent definition. This matter may be of theoretical interest, but does not concern us here.)

We will now define *selection attributes*, *routing attributes* and *control attributes*:

*Selection attributes* are defined to be those attributes that are true arguments to the trigger conditions.

$X_{ij}$ is a *selection attribute* if $X_{ij} \in arg(T(p))$ for some p

*Routing attributes* are those that are true arguments to some routing function (recall that routing functions are the components of an action $A(p)$ that modify the locations of the input messages).

$X_{ij}$ is a *routing attribute* if $X_{ij} \in arg(a_{k1})$ for some routing function $a_{k1}$.

*Control attributes* are attributes that are true arguments to any action that modifies some selection attribute, some routing attribute, or (recursively) some other control attribute:

$X_{ij}$ is a *control attribute* if:

(i) $X_{ij}$ is a selection attribute or

(ii) $X_{ij}$ is a routing attribute or

(iii) $X_{ij} \in arg(a_{kl})$ for some $a_{kl}$ and attribute $l$ of input $I_{pk}$ is a control attribute

*Routing* attributes are those that directly affect routing decisions. *Selection* attributes indirectly affect routing by determining which procedure is likely to "grab" the message (and consequently route it). *Control* attributes affect routing even more indirectly by influencing the value of routing or selection attributes. Note that the definition of *control attribute* is recursive, and so includes attributes that affect routing even indirectly.

Non-control attributes (the ones left over) do not influence routing or message flow in any way. Consequently we may ignore these when we decide how to partition our message state space. The non-control attributes are only of interest to us if we have specific questions about their value. We might, for example, like to know the range of values of a particular message field when it arrives at our station, even though that field in no way affects its flow through the network.

### 4.1.2. Trigger conditions

In our model we may express trigger conditions through the set $T(p)$ of acceptable input messages. This tells us nothing, however, about the structure of the condition. Let us consider some of the possibilities. The simplest trigger is no condition at all:

1. all messages of the correct type(s) are accepted

In this case all messages in the local scope of the procedure may trigger it. Since there is no other condition on the input, there can be no coordination between messages (*i.e.* if there is more than one message input, then there would be no trigger conditions comparing fields of one input to those of another). This sort of trigger is therefore likely to be useful only when there is a single message input. An example would be a procedure that automatically forwarded all *order* forms (figure 4.1) to another station.

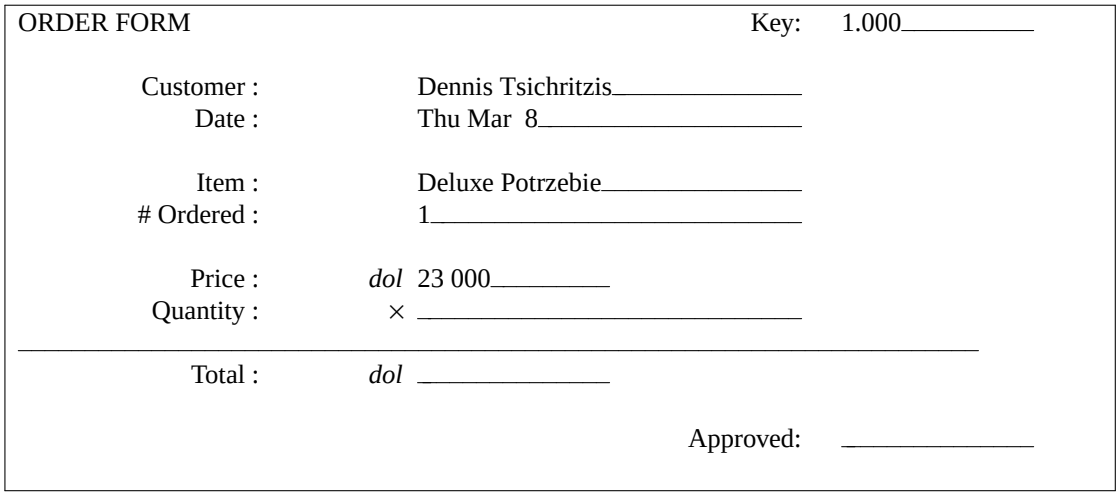



**Figure 4.1** : An order form

2. selections on attributes

We consider conditions in disjunctive normal form $\vee(\wedge C_j)$ where each $C_j$ is a simple condition comparing an attribute to some constant, that is $x_i \theta u$. Comparators such as $=, \neq, <, \leq, >, \geq$ may be used for numerical or text fields. Pattern matching in text and searching for voice patterns and bit maps in audio and visual fields is also conceivable. $x_i \sim$"Crete" might represent the condition that the text field $x_i$ contain the (constant) string "Crete". Low inventory items could be detected by a procedure that selects inventory records with a *quantity* less than 10, say, as in figure 4.2.

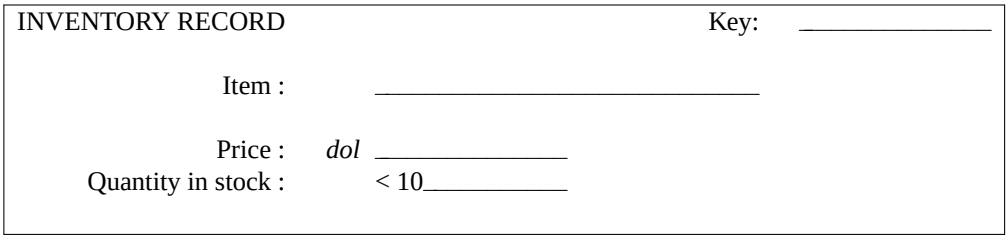



**Figure 4.2** : Low inventory trigger

3. predicates over message attributes

It may sometimes be necessary to select messages on a comparison of attributes. A procedure that selected orders that could not be completely filled would have to compare the *Quantity* field of an order form to the *# Ordered* field. Such conditions are of the form $x_i \theta x_j$. More complex conditions may be predicates over several attributes (for example, $x_i + x_j \leq x_k$).

4. joins between messages

Matching messages are identified by comparing similar attributes of different message inputs (usually of different message types). Although equality joins seem to be the most useful for matching messages, inequality joins are sometimes useful for identifying special cases of individual messages or pairs of matched messages. An inventory record and an order form may be matched by item name, for example, by applying an equality join. We may further select from amongst the message pairs retrieved by comparing the number in stock (on the inventory record) with the number ordered (on the order form) using an inequality join. We may thus identify orders that cannot be filled and (say) trigger a procedure that creates a backorder and requests more stock.

Arbitrarily complex conditions (involving more than one or two attributes) may exist in practice, but simple selections and joins are likely to be adequate for most applications. We shall therefore concentrate on these *without excluding* the possibility of other conditions. Such conditions may involve the evaluation of functions of several attributes, or the inclusion of information outside the system such as user input.

### 4.1.3. Actions

Actions modify messages and re-route them. The routing functions, $a_{k1}$ of an action $A(p)$ are constrained in that they must route messages to the finite set $R(s_i)$ (where $s_i$ is the location of $p$). If these routing functions are defined in terms of the input messages alone (and not user input or any other external source), then the input tuples $\tau \in T(p)$ can be partitioned into a finite number of independent subsets according to where the messages are routed. We may represent the set of tuples for which the $k$th message is sent to station $s_j$ by $\rho_{kp}(j)$:

$$\rho_{kp}(j) = \begin{cases} \{\tau \in T(p) |\ a_{k1}(\tau) = s_i\} & \text{if } j = 0 \\ \{\tau \in T(p) |\ a_{k1}(\tau) = m_{ij}\} & \text{if } 1 \leq j \leq N \\ \{\tau \in T(p) |\ a_{k1}(\tau) = \omega\} & \text{if } j = \omega \end{cases}$$

$\rho_{kp}(0)$ and $\rho_{kp}(\omega)$ are used to represent the case where message $k$ is not forwarded or is destroyed, respectively.

Because there is only a finite image space for routing functions, it is consequently possible to re-state these functions in the form:

$$a_{k1}(\tau) = \begin{cases} s_i & \text{if } \tau \in \rho_{kp}(0) \\ m_{i1} & \text{if } \tau \in \rho_{kp}(1) \\ m_{i2} & \text{if } \tau \in \rho_{kp}(2) \\ \cdots & \\ \omega & \text{if } \tau \in \rho_{kp}(\omega) \end{cases}$$

Furthermore, if we can express the $\rho_{kp}(j)$ "nicely" then we automatically have a corresponding definition for the routing functions. The earlier discussion of triggers may be applied here as well. If we can express $\rho_{kp}(j)$ in terms of simple conditions on attributes, then we may obtain routing functions that look like:

$$a_{k1}(\tau) = \begin{cases} s_i & \text{if } \forall(\wedge C_{0l}) \\ m_{i1} & \text{if } \forall(\wedge C_{1l}) \\ m_{i2} & \text{if } \forall(\wedge C_{2l}) \\ \cdots & \\ \omega & \text{if } \forall(\wedge C_{\omega l}) \end{cases}$$

If the conditions $C_{jl}$ compare attributes to constants, then these constants can be used to partition attribute domains into ranges.

The remaining components of an action may modify the attributes of the input messages. Each $a_{ij}$ potentially makes use of all the information available in all the input messages. In practice, however, not all attributes of a message will be modified. Most of the $a_{ij}$ will therefore be identity functions. Other simple actions may set attributes to constants. This is the case, for example, when a procedure automatically approves a request with a (constant) signature.

When actions set attributes to values that depend only on the previous values of those attributes, or on the values of other attributes in the same input message, then the next state of that message depends only on its previous state. If, however, information from the other inputs is needed to evaluate the new values of attributes, then we cannot determine the next state from the previous state alone. Furthermore, if external information (e.g. the date, user input) is required, then the actions are effectively non-deterministic (from the viewpoint of the model). The next state of a message can at best be determined as a set of possibilities.

Let us consider how actions may map a message from state to state. Much of what follows assumes numeric attribute domains, but often the arguments can be generalized to other domains, such as text fields.

Functions that depend on individual messages and map message fields to constants are the simplest to handle:

1. $a_{ij} \equiv \begin{cases} \text{if } \tau[i] \in \xi_1 \textit{ then set } x_j := u_1 \\ \text{if } \tau[i] \in \xi_2 \textit{ then set } x_j := u_2 \\ \dots \end{cases}$

   where $x$ is the $i$th input message, and each $\xi_k$ is a product of attribute ranges. Since attributes are set to constants, we can easily test whether they fall in the attribute ranges of any message state. Similarly, it is relatively straightforward to test what states $\sigma$ overlap a given $\xi_k$.

2. $a_{ij}$ is linear in $\tau$ or $\tau[i]$.

   This applies to numeric attributes. If $a_{ij}$ is a linear function of the inputs (i.e. a polynomial of degree 1), and we know what ranges of values the arguments may assume, then the image of $a_{ij}$ can also be expressed as a range. If, for example, $a_{ij}$ is defined to be $x_k + y_l$ (where $x$ and $y$ are input messages in $\tau$), and we know that $x_k \in [10, 20]$ and $y_l \in [4, 6]$, then we can deduce that the image must lie in the range $[14, 26]$.

3. $a_{ij}$ is polynomial in $\tau$ or $\tau[i]$.

   A function that computes yearly interest compounded monthly would be a twelfth-degree polynomial. In this case, we may have difficulty telling what the image of a given message state will be. Finding the minimum and maximum of $\{ f(x) | x \in \prod R_k \}$ cannot be solved exactly for a polynomial $f(x)$ of degree higher than four (since we must find the roots of its derivative). With a bit of work, reasonable bounds may be found, however.

4. $a_{ij}$ is *monotone*.

   Any mapping that is monotone increasing or decreasing maps ranges neatly to ranges. In this case we can be sure that $a_{ij}$ attains its minimum and maximum at some "corner" of a message state (where the "corners" correspond to the attributes assuming the extreme values of the attribute ranges). Many polynomials will be monotone over the domain of concern, such as the compound interest example above.

5. $a_{ij}$ is arbitrary.

   If actions are not "well-behaved", then there is little hope of recovering any useful information about modified fields. If $a_{ij}$ behaves especially badly, it may map elements of one message state to all other states. This is the case, for example, when the action is an arbitrary modification of the message by the user. Since we cannot predict what changes will be made (assuming the modification is without restriction), we have no way of limiting the possible next states.

## 4.2. Analyzing message flow

We have thus far introduced the notion of a "message state", we have defined "control attributes", and we have classified several kinds of trigger conditions and actions. In this section we shall develop a method for partitioning message domains into state spaces and some techniques for obtaining message paths in terms of message states.

The first step is to identify the control attributes, since they are the ones that influence message routing. We present a distributed technique for collecting this information from all the workstations in the system. The next task is to extract the information in the trigger conditions and actions that will tell us how to partition the control attribute domains. This will yield our message state space.

We can then translate trigger conditions and actions into operations on message states. We thus obtain a finite-state automaton representation of message flow, with one automaton per message type. The state transitions for the automata can also be gathered in a distributed fashion by the use of "symbolic messages". The symbolic messages represent messages in various states, and they travel from station to station, logging the path they trace. Since messages may be routed in different directions depending on coordinating messages they encounter, the symbolic messages may "split" along the way. The children of a symbolic message eventually return to the originating station, and, through the use of "splitting histories", the information gathered is reconstructed.

### 4.2.1. Detecting control attributes

We have previously defined control attributes as those attributes that affect routing directly or indirectly. This includes selection attributes that are examined by trigger conditions, routing attributes that are used to compute the next location of a message, and (recursively) any attribute that is used to compute the next value of any other control attribute.

We shall present a distributed algorithm for collecting the control attributes present in the system. Briefly, whenever new procedures are created or old ones modified, a station locally detects the new control attributes. These are then broadcast to all other stations. If yet more control attributes are discovered at the other stations, then these too are broadcast. Every receipt of new control attributes must be acknowledged with a message telling whether more are discovered or not. Since there are only a finite number of attributes, this procedure must eventually terminate.

Discovering what the control attributes are for a given message type is not inherently difficult. We assume that it is fairly easy to tell what the arguments to a trigger condition or an action are. Routing attributes and selection attributes are then trivial to detect. To discover the remaining control attributes one need only apply the recursive definition until no more attributes are found. The only real twist is that we wish to know the *global* control attributes, that is, we are concerned with all procedures at all sites, not just those at a single station. When all stations are on the same physical machine, then this may not pose any special problems, but it is far more reasonable to assume that our stations are on separate physical machines connected by a network.

Although it may be possible to collect all the information concerning procedure arguments at a single station that does all the processing to determine the control attributes, it is desirable to have the option of running such an algorithm in a distributed fashion. Where all stations are equal, we may well prefer not to unnecessarily burden one with the "dirty work" of analyzing system behaviour.

We introduce the notation:

$$\gamma(X_{ij},\ p) = \bigcup \{arg(a_{kj}) |\ I_{pk} = X_i\}$$

to represent the set of attributes that affect the computation of any action in procedure $p$ that modifies attribute $X_{ij}$. Clearly, if $X_{ij}$ is a control attribute, then so are any attributes in $\gamma(X_{ij}, p)$. Of course, $\gamma(X_{ij}, p)$ is empty if $X_i \notin I(p)$. We extend $\gamma$ to sets of attributes and sets of procedures in the obvious way.

In our distributed algorithm each station is responsible for detecting locally all routing attributes and control attributes. This is done whenever a procedure is added. New control attributes are then broadcast to all other stations. Stations receiving new control attributes then apply $\gamma$ recursively to detect any further

control attributes. If more are discovered, then these too are broadcast. The algorithm terminates when no station has anything left to broadcast. We can detect termination by insisting that all stations acknowledge broadcasts by stating whether or not any new control attributes result.

Each station maintains $3 \times N$ lists:

$$\Lambda_i, \Delta_i \text{ and } \Gamma_i$$

Each $\Lambda_i$ is the list of control attributes for message type $X_i \in X$. Initially each $\Lambda_i$ is empty. As new procedures are added to the system, these lists are updated to include newly-discovered control attributes.

$\Delta_i$ and $\Gamma_i$ are temporary lists for keeping track of new control attributes of $X_i$, as they are discovered. The $\Gamma_i$ are the pending lists of control attributes to be broadcast. In addition, each station maintains a list *ack* of broadcasts awaiting acknowledgement. *ack*, when it is not empty, contains tuples of the form:

$$(s, n, l)$$

where $s$ is the station initiating the broadcast, $n$ is a unique broadcast sequence number for $s$, and $l$ is the list of stations that have acknowledged the broadcast. $l$ is always initialized to contain the broadcasting station $s$. We assume that every station knows what other stations exist. (Note that $l$ could be replaced by a counter that is used to simply keep track of the number of acknowledgements received.) We also assume that the network can reliably handle 'broadcasting' even though all stations may not necessarily be up at the same time.

When a station $s$ broadcasts control attributes, it sends a message to all other stations of the form:

$$NEW(s, n, \Gamma_1, \ldots, \Gamma_N)$$

where the $\Gamma_i$ contain new control attributes, *and* at least one $\Gamma_i$ is non-empty. $n$ is a unique broadcast number for $s$.

When station $s$ receives a *NEW* broadcast, $NEW(s', n', \cdots)$, from some other station $s'$, it must in turn broadcast an acknowledgement of the form:

$$ACK(s, s', n', \varnothing)$$

if there are no new control attributes, or

$$ACK(s, s', n', NEW(s, n, \Gamma_1, \ldots, \Gamma_N))$$

if there are.

The algorithm has several parts. Each part is run independently at every station, when required. Each station must ensure that parts *A*, *B* and *C* are *not* run concurrently (since they access the same data structures) but if more than one needs to be run at any time, they may be executed in any order. *A* generates *NEW* broadcasts, *B* processes *NEW* broadcasts and acknowledges them with *ACK* broadcasts, and *C* processes *ACK* broadcasts. When we refer to a "new" control attribute, we mean one that is not to be found in $\Lambda_i$, $\Delta_i$ or $\Gamma_i$. The following is to be run whenever a new procedure $p$ is created:

A1. add all new routing attributes and selection attributes $X_{ij}$ of $p$ to $\Delta_i$
A2. add all new control attributes $X_{ij}$ in $\gamma(\Lambda_i, p)$ to $\Delta_i$
A3. **if** every $\Delta_i$ is empty **then** STOP
**else** {
A4. **for each** $\Delta_i$ and $X_{ij} \in \Delta_i$ **do** {
A5. move $X_{ij}$ from $\Delta_i$ to $\Gamma_i$
A6. add each new $X_{kl} \in \gamma(X_{ij}, P(s))$ to $\Delta_k$
(continue until every $\Delta_i$ is empty)
}
A7. broadcast the non-empty $\Gamma_i$ with $NEW(s, n, \Gamma_1, \ldots, \Gamma_N)$ and make a broadcast entry $(s, n, \{s\})$ in *ack*
A8. move all $X_{ij} \in \Gamma_i$ to $\Lambda_i$
}
STOP

New control attributes can result from procedures being created or from processing a broadcast from another station. Procedure *B* is run by stations receiving *NEW* broadcasts. It is very similar to *A* except for the acknowledgement that must be generated. When station $s$ receives a *NEW* broadcast $NEW(s', n', \cdots)$, from station $s'$, it must be processed as follows:

B1. make a broadcast entry $(s', n', \{s, s'\})$ in *ack*
and add new control attributes to $\Gamma_i$
B2. **for each** $\Gamma_i$ and $X_{ij}$ in $\Gamma_i$ **do** {
B3. move $X_{ij}$ to $\Lambda_i$
B4. add each new $X_{kl} \in \gamma(X_{ij}, P(s))$ to $\Delta_k$
(continue until every $\Gamma_i$ is empty)
}
B5. **if** every $\Delta_i$ is empty **then** {
B6. acknowledge receipt with no new attributes resulting i.e. broadcast $ACK(s, s', n', \varnothing)$
}
**else** {
B7. **for each** $\Delta_i$ and $X_{ij} \in \Delta_i$ **do** {
B8. move $X_{ij}$ from $\Delta_i$ to $\Gamma_i$
B9. add each new $X_{kl} \in \gamma(X_{ij}, P(s))$ to $\Delta_k$
(continue until every $\Delta_i$ is empty)
}
B10. acknowledge receipt and broadcast the non-empty $\Gamma_i$
i.e. broadcast $ACK(s, s', n', NEW(s, n, \Gamma_1, \ldots, \Gamma_N))$
make a broadcast entry $(s, n, \{s\})$ in *ack*
B11. move all $X_{ij} \in \Gamma_i$ to $\Lambda_i$
}
STOP

Every *NEW* broadcast of control attributes that is received must result in an acknowledgement, regardless of whether more control attributes are discovered. Acknowledgements may or may not be accompanied by further broadcasts. When an acknowledgement of the form $ACK(s', s'', n'', \varnothing)$ or $ACK(s', s'', n'', NEW(s', n, \Gamma_1, \ldots, \Gamma_N))$ is received by station $s$, *ack* must be updated:

C1. update $(s'', n'', l) \in ack$ to $(s'', n'', l \bigcup \{s'\})$
C2. **if** that entry is complete **then** delete it
C3. **if** *ack* is empty **then** all control attributes are known
C4. **if** the acknowledgement is of the second form, invoke B on $NEW(s', n, \Gamma_1, \ldots, \Gamma_N)$

Parts A and B guarantee that all routing and control attributes will be known to all stations. Part A recursively applies $\gamma$ to the new routing and selection attributes to detect any more new control attributes. Step A2 is needed in case any new control attributes arise from the new procedure $p$ modifying old control

attributes.

In part B we first check newly-arrived control attributes to see if they locally yield any more (steps B1 through B4). If they do not, then we acknowledge receipt, and we are done. If they do, then we must apply $\gamma$ recursively to obtain all of them, and then broadcast the lot (steps B7 through B11). Note that we must apply $\gamma$ to the new control attributes in two steps, since we need not re-broadcast the new arrivals. We only broadcast new control attributes if any result from the first application of $\gamma$.

Every broadcast must be acknowledged by all other stations. When all broadcasts have been acknowledged with no new broadcasts resulting, then we know that every station has the same collection of $\Lambda_i$ and all control attributes are known.

Since there are only a finite number of attributes to begin with, the algorithm must eventually terminate. Furthermore, note that in steps A1, A2 and A6, and also in steps B4 and B9, we only investigate triggers or actions that have not been looked at before. (By "actions" we mean the individual $a_{ij}$s of a procedure.) This means that the algorithm checks each trigger and each action at most once. Whenever a new procedure is added, the amount of work to be done is linearly bounded by the number of attributes that have not yet been identified as control attributes plus the number of triggers and actions in all procedures that have not yet been checked (i.e. those of the new procedure, and those actions of existing procedures that modify non-control attributes).

### 4.2.2. Obtaining message states

We will now consider the matter of how best to partition message domains into state spaces. Simple trigger conditions provide us with excellent partitions, but complex conditions yield unusual message subdomains whose images under actions can be hard to follow. Since we are interested especially in the effect of actions on message states, it is important to have states that are as simple as possible to trace. We may therefore try to "box" complex subdomains, or reduce a complex condition to a collection of simple conditions that cover it. We may also try to refine our partition by discovering new message states that result from applying actions to existing message states. This "fine-tuning" may be continued indefinitely, however, and so it is generally not practical to carry it too far.

Generally speaking, the best message state space would identify one message state per message value. Since we require a finite number of message states to begin to analyze message flow, we must consider carefully how we choose our partition.

Since control attributes are the only attributes that affect routing, our message states should correspond to predicates over the control attributes. We can gather this information at the same time that we collect the control attributes in the above algorithm.

Selection attributes are those that are arguments to trigger conditions. The trigger conditions thus automatically yield conditions that may be usable for generating message states. If a trigger condition can be expressed as $\vee(\wedge C_j)$ where each $C_j$ is a predicate involving one or more control attributes, then we can use the $C_j$ to generate message states. The conditions collected in this way at all stations yield a state space by considering messages that may or may not satisfy each of these conditions. If, for example, there are $c$ conditions in total that involve messages of type $X_i$, then a message $x \in dom(X_i)$ may potentially fall in one of $2^c$ message states, corresponding to success or failure in matching each of these conditions.

Of course, not all combinations of conditions necessarily yield a usable message state: some combinations may be contradictory. Conditions $x_i > 5$ and $x_i < 3$ clearly cannot both be true at the same time. There may therefore be considerably less than $2^c$ non-empty message states.

We have previously mentioned the desirability of message states expressible as a product of ranges or attribute subdomains. Message states that are expressible as a Cartesian product of attribute subdomains allow us to consider each attribute independently. We would thus have

$$\sigma = \prod_{j=0}^{n_i} R_j$$

or

$$\sigma = \{ x \in dom(X_i) | \bigwedge_j C_j \}$$

where each $C_j$ represents $R_j$. $C_j$ is therefore a simple condition involving only attribute $X_{ij}$, for example: $4 \leq x_j \leq 10$.

If the trigger conditions $\bigvee(\bigwedge C_j)$ have the property that each $C_j$ is a simple condition of this form, then we automatically are able to derive our desired message states. Furthermore, when the attributes are numeric and the conditions are of the form $x_i \theta u$ where $u$ is a constant and $\theta \in \{ =, \neq, <, \leq, >, \geq \}$ then the conditions yield attribute ranges bounded by the constants. In this case, if we have $c_j$ conditions involving attribute $X_{ij}$, we have at most $c_j$ constants and at most $c_j + 1$ ranges. Consequently we would have $\prod_j (c_j + 1)$ message states (where $c_j = 0$ for non-control attributes). This is considerably less than the potential $2^c$ states resulting from non-simple conditions (where $c$ is the total number of conditions involving all $X_{ij}$, i.e. $c = \sum c_j$).

Unfortunately we cannot reasonably expect all trigger conditions to be this well-behaved. There are two options available. The first is to ignore all $C_j$ that are not of the form $x_i \theta u$, and the other alternative is to try to convert them to simpler conditions that are more useful. The idea is to "box" the messages satisfying the condition by discovering the attribute ranges that correspond to solutions of the predicate. This can be done, for example, with a condition like:

$$x_i^2 + x_j^2 \leq 25$$

Here we can deduce that $-5 \leq x_i \leq 5$ and $-5 \leq x_j \leq 5$. With the condition:

$$x_i = x_j$$

however, we can deduce nothing since both attributes potentially range over their entire domains. Note that we may use combinations of conditions to extract more information. If, for example, the condition above were combined with $x_j > 0$, then we may deduce that $x_i > 0$ is also of interest. In a trigger condition of the form $\bigvee(\bigwedge C_j)$, one should use the conjunctions $\bigwedge C_j$ to deduce the simple conditions.

We may take the same approach with routing attributes. Earlier in this chapter we introduced the notation $\rho_{kp}(j)$ to represent the input tuples for which the $k$th message is routed to station $s_j$. We may similarly "box" each $\rho_{kp}(j)$ to obtain simple conditions on control attributes.

In the cases of both selection attributes and routing attributes, the problem is greatly simplified if triggers and routing actions are expressed by users in terms of fairly simple conditions on attributes. Furthermore, the user may be asked to supply any additional information implied by conditions that involve comparisons of several attributes. Of course, depending on the complexity of the triggers and actions expressible within the system, it would be desirable if the system itself could do all the analysis of attribute ranges.

Other control attributes are slightly more complicated to handle since they appear in actions that may not map to finite sets. We have, however, already obtained ranges for the control attributes found thus far (the routing and selection attributes), so we may feel free to use this information at this point.

Consider a control attribute $X_{ij}$ that is modified by $a_{kj}$ of procedure $p$ (where $X_i = I_{pk}$). By the definition of "control attribute", we know that all attributes in $arg(a_{kj})$ must also be control attributes. Also, since $X_{ij}$ is a control attribute already discovered, we presumably have some range information about it. If $R_l$ is a range for $X_{ij}$, then:

$$a_{kj}(\tau) \in R_l$$

is a predicate over the inputs $\tau$ to procedure $p$. We may therefore attempt to "box" the set of inputs that satisfy this condition, and thereby obtain ranges for the control attributes in $arg(a_{kj})$. The new ranges can be used to further subdivide, or "fine-tune" the message states.

Note again that "boxing" may be impossible in some cases, yet trivial in others. Specifically, if $a_{kj}$ is a function of a single argument, then the condition $a_{kj}(\tau) \in R_l$ is a predicate over a single attribute. For

example, if $a_{kj}$ returns something like $x_h + 1$, and $R_l$ is the range $[a, b]$, then the resulting predicate is $x_h + 1 \in [a, b]$, and the resulting range for this attribute will (trivially) be $[a - 1, b - 1]$.

If, on the other hand, $a_{kj}$ is a complicated function of several arguments (for example, a high-order polynomial), then the task of obtaining attribute ranges is a problem in numerical analysis with only approximate solutions available.

We have, therefore, techniques for extracting attribute ranges (or subdomains) for selection attributes, routing attributes, and all other control attributes. Whenever a new control attribute is discovered in the algorithm of the previous section (in steps A1, A2, A6, B1, B4 and B9), we may determine the ranges at the same time. New range information about existing control attributes can also be detected. This information may be broadcast at the same time.

Special care must be taken at one phase, however. When we analyze actions, we use the known ranges of other control attributes. More ranges may be discovered, and one may be tempted to fine-tune the analysis. Consider the following trivial example to see what may happen: Suppose $x_i$ is a selection attribute in the trigger condition $5 \le x_i \le 10$. We are therefore interested in the range $[5, 10]$. Suppose there is an action that sets $x_i := x_i - 1$. If we analyze this we discover that we are also interested when $x_i - 1 \in [5, 10]$, i.e. the range $[6, 11]$. If we continued recursively we would never stop finding new ranges.

Since this problem only occurs in the recursive step, an obvious solution is to prohibit fine-tuning of existing control attributes appearing in actions. Selection attributes and routing attributes do not present any problems.

### 4.2.3. State transitions

At this point in our analysis we expect each station to know what message states are currently of interest. What is left is to determine what state transitions are effected by the procedures. For a message in a given input state $\sigma$ we would like to know the possible next states, $\sigma'$ that may result if the message triggers some procedure $p$.

To tell what happens when $p$ fires, it is not, in general, sufficient to know the state of a single input message. Attributes of all coordinating messages are potentially available to the actions that modify the message we are interested in. Although we cannot predict what states the other inputs will be in, we know that they must satisfy the trigger condition. We therefore introduce the following notation to represent the possible inputs given one message in state $\sigma$:

$$\tau_p(\sigma) = \{\tau |\ \tau \in T(p),\ \tau[k] \in \sigma\}$$
(where $\sigma \subseteq dom(X_i)$ and $X_i = I_{pk}$)

(For simplicity, $X_i$ and $k$ are understood.) Note that $\tau_p(\sigma)[k]$ is the set of message values in $\sigma$ that may trigger $p$ (possibly empty). This is equal to $\sigma \cap T(p)[k]$.

Recall that $\hat{p}(x)$ was the set of procedures that $x$ might trigger, and $\hat{a}_p(x)$ was the set of values that $x$ might be mapped to after triggering $p$. We extend our definitions of $\hat{a}$ and $\hat{p}$ from chapter 3 to message states:

$$\hat{p}(\sigma) = \{p \in P |\ \tau_p(\sigma) \neq \emptyset\}$$

$$\hat{a}_p(\sigma) = \{A(p)(\tau)[k] |\ p \in \hat{p}(\sigma),\ \tau \in \tau_p(\sigma),\ X_i = I_{pk}\}$$

Procedure $p$ then effects a state transition from $\sigma$ to $\sigma'$ if $p \in \hat{p}(\sigma)$ and $\hat{a}_p(\sigma) \cap \sigma' \neq \emptyset$. That is $p: \sigma \to \sigma'$ if $p$ is capable of mapping some message in state $\sigma$ to some message in state $\sigma'$, given the right coordinating messages. We may extend $\hat{l}$ from chapter 3 to apply to message states as well:

$$\hat{l}(\sigma) = \begin{cases} \{p\hat{l}(\sigma') | p \in \hat{p}(\sigma),\ \hat{a}_p(\sigma) \cap \sigma' \neq \emptyset\} & \text{if } \sigma \neq \omega \text{ and } \hat{p}(\sigma) \neq \emptyset \\ \lambda \ \text{(the empty string)} & \text{otherwise} \end{cases}$$

$\hat{l}(\sigma)$ therefore is the message flow language for message state $\sigma$. It represents all sequences of procedures that messages in state $\sigma$ may possibly encounter. $\hat{l}(\sigma)$ may be "computed" by recursively applying its definition. Sequences of procedures are generated as $\hat{l}(\sigma)$ is expanded. (Of course, a straightforward expansion is impractical since infinite strings may be generated.)

Since messages in different states may still be able to trigger the same procedures, it is useful to keep track of the message states together with the sequences of procedures encountered. We spoke earlier of a message path as an alternating sequence of message values and procedures. We may easily extend this idea to message states in the following definition:

$$\phi(\sigma) = \begin{cases} \{\sigma\, p\phi(\sigma') | p \in \hat{p}(\sigma),\ \hat{a}_p(\sigma) \bigcap \sigma' \neq \varnothing\} & \text{if } \sigma \neq \omega \text{ and } \hat{p}(\sigma) \neq \varnothing \\ \sigma & \text{otherwise} \end{cases}$$

Note the similarity to the definition of $\hat{l}$. In fact, we may obtain $\hat{l}(\sigma)$ by mapping the states in $\phi(\sigma)$ to the empty string. $\phi(\alpha)$ represents paths starting from message creation. Paths terminate when messages are destroyed, so $\phi(\omega) = \omega$.

At this point we can easily see that message behaviour can be compared to that of a finite state automaton. Let $\Sigma_i$ be the set of message states for message type $X_i$, i.e. $\Sigma_i$ is a partition of $dom(X_i)$ obtained by the approach described in the previous section. Then the finite automaton of $X_i$ is:

$$< \Sigma_i,\ P \times \Sigma_i,\ \delta_i,\ \alpha,\ \omega >$$

The states of the automaton are the message states. Inputs are strings over $P \times \Sigma_i$, i.e. pairs of procedures and next-states. The initial state is $\alpha$, the final state $\omega$, and the next-state function is:

$$\delta_i(\sigma,\ (p,\ \sigma')) \mapsto \sigma'$$

where $X_i = I_{pk},\ p \in \hat{p}(\sigma)$ and $\hat{a}_p(\sigma) \bigcap \sigma' \neq \varnothing$. Note that we have $K$ automata, one for each message type. We shall discuss how these automata can been seen to interact in chapter 5.

Determining what the state transitions are may not be so easy. One of the difficulties may lie in evaluating $\hat{a}_p(\sigma)$. In the earlier section on actions we enumerated some of the possible functions that may be encountered in procedures. If we assume that the states in $\Sigma_i$ are all products of attribute ranges, then functions that are monotone (linear, monotone polynomial, etc.) are relatively easy to analyze. With monotone functions we need only consider the allowable ranges of the argument attributes to determine the output ranges. Consider the function $x_i + x_j$. If in the input message state we have $x_i \in [0, 5]$ and $x_j \in [10, 20]$ then the image of the function $x_i + x_j$ for that message state is the range $[10, 25]$. If this function sets field $x_k$, then we need only determine what attribute ranges of that field intersect $[10, 25]$. With non-monotone functions we have the "boxing" problem mentioned in the previous section.

Functions whose image is a finite domain may be stated in the form:

$$a_{ij}(\tau) = \begin{cases} u_1 & \text{if } C_1 \\ u_2 & \text{if } C_2 \\ \cdots & \end{cases}$$

where each $u_k$ is a constant and each $C_k$ is some condition on the input $\tau$. For a given message state $\sigma$ we must therefore determine which $C_k$ are satisfied by the inputs $\tau \in \tau_p(\sigma)$. We assume that such functions are available to us in this form, or that the $C_k$ can at least be derived. If the $C_k$ are composed of simple conditions of the form $x_i \theta u$, where $\theta$ is an inequality, then the evaluation is straightforward. Perverse conditions like $x_3^7 - x_5 + \dfrac{x_7}{x_8^3 + x_8} = 0$ cause problems because we must solve them over the domain $\tau_p(\sigma)$ to tell what the next state may be.

Clearly, it is possible to devise actions that are miserably tricky to analyze. There are several possible ways of dealing with this. One approach is to design the system so that only "nice" actions may be

used. Another possibility is to have the system request information about new functions such as whether they are monotone, or, more specifically, how they are expected to map message states (if the states are known).

One interesting approach may be to have the system do its best to determine what the state transitions are, and then monitor *actual* message state transitions to see if they agree with its analysis. For every message we must determine what message state it belongs to, and when it is modified by a procedure, check what its next-state is. If the transition is one that was predicted, then we know all is well. If not, then the transition must be added to the known set, and the discrepancy can be pointed out to a system programmer.

The set of all state transitions can be found by having each station determine what transitions may occur there. Not all message states may be reachable, however. (Similarly, not all state transitions are "reachable".) An alternative way of finding the state transitions is to start with the procedures that are capable of creating new messages, and to trace message state transitions starting from there. The reachable state transitions are thus collected by following the paths in $\phi(\alpha)$. Since there are only a finite number of transitions, an algorithm to compute $\phi(\alpha)$ should terminate after encountering each transition at most once. We shall investigate such an algorithm in the following section.

### 4.2.4. Symbolic messages

We collect reachable state transitions by using a *symbolic message* that represents a choice of possible current message states and keeps track of the transitions that have been traversed up to that point. Since different messages are often routed in different directions by procedures, we need the ability to *split* a symbolic message whenever this happens. A symbolic message may thus split into many parts going in different directions before all reachable states and all state transitions are found.

A symbolic message gathers all the reachable state transitions by simply traversing a "spanning tree", starting at $\alpha$, and visiting each station where the information about the transitions resides. (A *spanning tree* of a graph is a subgraph of that graph that both covers all the vertices of that graph, and is a tree [BoMu76].) When there are no new states and state transitions to visit, it returns to the station initiating it. Since the symbolic message may have split into separate parts, the work is not finished until each of the parts returns. When the transitions have all been gathered, we may then generate a regular expression capturing the message flow automaton by using a standard algorithm such as in [AhHU74]. (These algorithms have complexity $O(n^3)$, where $n$ is the number of states in the automaton.)

Consider the message flow automaton in figure 4.3. Here messages are created by procedure $p_1$ and are destroyed by procedure $p_5$.

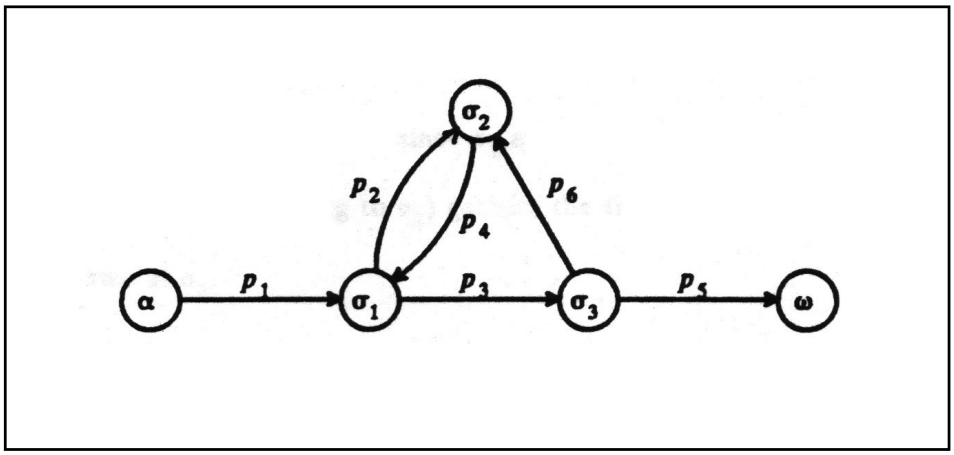



**Figure 4.3** : A message flow automaton

Suppose that each state corresponds to a different station, so that the location of $\sigma_1$ is $s_1$, and so on. Then the procedures in the figure belong to the stations as follows:

$$p_1 \in P(s_0)$$
$$p_2,\ p_3 \in P(s_1)$$
$$p_4 \in P(s_2)$$
$$p_5,\ p_6 \in P(s_3)$$

Each procedure thus sends the message to a new station, after modifying it.

A symbolic message would start at state $\alpha$ and follow transition $p: \alpha \mapsto \sigma_1$. At $\sigma_1$ it would split into two directions and follow $p_2$ and $p_3$ to $\sigma_2$ and $\sigma_3$, respectively. From $\sigma_2$ it could only go back to $\sigma_1$, which has been seen already. From $\sigma_3$, it could go to $\sigma_2$ and $\omega$. This causes another split. $\sigma_2$ has also been seen already, and there is nowhere to go from $\omega$, so the symbolic messages would terminate, having traversed all possible state transitions.

Note that we may have multiple paths to a single message state $\sigma$. There are multiple paths, for instance, to state $\sigma_2$ in our example. Different children of the original symbolic message may encounter this state at different times, but, of course, only the first one to arrive is needed to continue and compute $\phi(\sigma)$. The other children need to know that the first one has already been there, so, in addition to the information kept by the symbolic messages, we must keep track at each station which states have been reached thus far by some symbolic message. The children in our example result from the splits at $\sigma_1$ and $\sigma_3$.

The first split occurs at $\sigma_1$, since $p_2$ and $p_3$ route the message to different locations. The first child of that split (going to $\sigma_2$) gathers the transitions from $\sigma_2$. The second child is itself split into two at $\sigma_3$.

The first grandchild terminates at $\omega$. The remaining grandchild, follows $p_6$ to $\sigma_2$. There it terminates, since state $\sigma_2$ has already been visited by the first child of the first split. (That information must be maintained at the station that $\sigma_2$ belongs to.)

Finally, we need to know when all the symbolic messages have finished their work. This means we need to somehow keep track of how many descendants there are of the original symbolic message. One way to do this is to note that we can visualize a tree of symbolic messages with the original one as the root. Let us label each edge of the tree with a pair $(j, n)$, where edge $(j, n)$ is the $j$th of $n$ edges leaving a node. If the tree grows downward, then the leftmost edge is always labeled $(1, n)$, and the rightmost $(n, n)$.

Now we may label each node of the tree with the sequence of edge labels along the path from the root of the tree to that node. Since each edge incident with a given node is uniquely identified for that node, the sequence of edge labels uniquely identifies each node in the entire tree. The leftmost leaf node therefore has the label $(1, m) \cdots (1, n)$, and the rightmost leaf node has the label $(m, m) \cdots (n, n)$. We call such a sequence of pairs a *splitting history*. The splitting history (1,4)(2,2) identifies the second child of the first child of the root. It also indicates where other children have split off. By comparing the splitting histories of the symbolic messages, we can tell whether they have all returned or not.

A symbolic message, then, is a tuple:

$$\nu = (s,\ \Theta,\ \Sigma,\ \chi)$$

where $s$ is the originating station, $\Theta$ is the collection of state transitions gathered thus far, $\Sigma$ is the set of message states currently represented, and $\chi$ is a splitting history. (Additional information such as the message type of the symbolic message and a unique identifier may also be needed since there will typically be many such symbolic messages floating through the system. For simplicity's sake we shall take this for granted.)

The state transitions in $\Theta$ are represented by triples,

$$(p,\ \sigma,\ \sigma') \text{ such that } p \in \hat{p}(\sigma) \text{ and } \hat{\alpha}(\sigma) \bigcap \sigma' \neq \varnothing$$

$(p, \sigma, \sigma')$ is therefore a transition from state $\sigma$ to state $\sigma'$.

Each station must also maintain a list *SEEN* to keep track of the states that have been seen by a descendent of some given original symbolic message. (Again, we need a unique set *SEEN* for every given original symbolic message and its descendents. For simplicity we shall take this as understood.)

Briefly, at each station that a symbolic message visits, we must add new transitions to $\Theta$, determine the new states in $\Sigma$ that the symbolic message represents, and split into several new symbolic messages, if necessary.

If station $s$ has a procedure $p$ that creates messages of type $X_i$ then it may initiate a symbolic message $v$ with $\Theta = \emptyset$, $\Sigma = \{\alpha\}$ and $\chi = \lambda$, the empty string. As $v$'s children arrive at each station in the system, the following steps must be taken: ($\Sigma'$ and $\Sigma_1, \ldots, \Sigma_n$ are temporary variables.)

D1. **for each** $\sigma \in \Sigma$ **do** {
D2. **if** $\sigma \notin SEEN$ **and** $\hat{p}(\sigma(\pi)) \neq \emptyset$ **then** {
D3. **for each** $p \in \hat{p}(\sigma)$ and $\sigma'$ such that $\hat{a}_p(\sigma) \cap \sigma' \neq \emptyset$ **do** {
D4. add $(p, \sigma, \sigma')$ to $\Theta$
D5. add $\sigma'$ to $\Sigma'$
($\sigma'$ may be $\omega$)
}
D6. add $\sigma$ to *SEEN*
}
}
D7. replace $\Sigma$ by $\Sigma'$
D8. partition $\Sigma$ into $\Sigma_1, \ldots, \Sigma_n$ so that every state $\sigma \in \Sigma_j$ is at the same station
D9. **if** $n > 1$ **then do** {
D10. split $v$ into $v_1, \ldots, v_n$ such that:
D11. $v_j = (s, \Theta_j, \Sigma_j, \chi_j)$
D12. for $j = 1$, $\Theta_j = \Theta$
D13. for $j > 1$, $\Theta_j = \emptyset$
D14. $\chi_j = \chi(j, n)$
}
D15. **for each** $v_j$ **do** {
D16. **if** $\Sigma_j = \emptyset$ or $\Sigma_j = \{\omega\}$ **then**
D17. send $v_j$ back to $s$
D18. **else** send $v_j$ to the station of $\Sigma_j$
}
STOP

In our example, we start out with:

$$(s_0, \emptyset, \{\alpha\}, \lambda)$$

As we traverse the transitions, we obtain:

$$(s_0, \emptyset, \{\alpha\}, \lambda)$$
$$(s_0, \{(p_1, \alpha, \sigma_1)\}, \{\sigma_1\}, \lambda)$$
$$(s_0, \{(p_1, \alpha, \sigma_1), (p_2, \sigma_1, \sigma_2), (p_3, \sigma_1, \sigma_3)\}, \{\sigma_2, \sigma_3\}, \lambda)$$

At this point we split into:

$$(s_0, \{(p_1, \alpha, \sigma_1), (p_2, \sigma_1, \sigma_2), (p_3, \sigma_1, \sigma_3)\}, \{\sigma_2\}, (1, 2))$$

and

$$(s_0, \emptyset, \{\sigma_3\}, (2, 2))$$

(Note that only the first child keeps the transitions gathered thus far.) The former yields:

$$(s_0, \{(p_1, \alpha, \sigma_1), (p_2, \sigma_1, \sigma_2), (p_3, \sigma_1, \sigma_3), (p_4, \sigma_2, \sigma_1)\}, \{\sigma_1\}, (1, 2))$$

and the latter yields:

$$(s_0, \{(p_5, \sigma_3, \omega), (p_6, \sigma_3, \sigma_2)\}, \{\omega, \sigma_2\}, (2, 2))$$

At station $s_3$ we cannot tell that state $\sigma_2$ has been reached already, so we split again into:

$$(s_0, \{(p_5, \sigma_3, \omega), (p_6, \sigma_3, \sigma_2)\}, \{\omega\}, (2, 2)(1, 2))$$

and

$$(s_0, \varnothing, \{\sigma_2\}, (2, 2)(2, 2))$$

Both children then die, the first from reaching $\omega$, and the second from reaching a state that has been seen before.

Each symbolic message always represents only messages that encounter the same stations. Whenever messages may be routed in different directions, a symbolic message must "split". The children are then routed to their new locations. Children are routed back to the originating station $s$ for any of four reasons:

1. $s$ appears normally in the message path
2. messages are destroyed; the final state $\omega$ has been reached
3. messages have reached a dead end; there are no procedures to handle those message states
4. the message state encountered has been seen by another symbolic message; to continue would unnecessarily duplicate work

Only the last three cases mean that the symbolic message has finished its work.

When symbolic messages return to the originating $s$, the $\chi$ may be examined to determine whether there are more children yet to arrive. Simply keep all the $\chi$ in a sorted list, such that

$$\chi < \chi' \text{ if } \chi = a(j, n)b,\ \chi' = a(k, n)c \text{ and } j < k$$

where $a$ and $b$ are (possibly empty) sequences of pairs. $\chi$ and $\chi'$ will have split after the common sequence $a$. When there are no "gaps" left in the list, we are done. To detect gaps, we need the following definitions:

i) A sequence $b$ is "*initial*" if it is of the form $(1, m), \ldots, (1, n)$. That is, the first element of each pair in the sequence is a 1. An empty sequence is always initial. An initial sequence identifies the leftmost child of some branch of a tree (the tree grows "downward").

ii) A sequence $b$ is "*complete*" if it is of the form $(m, m), \ldots, (n, n)$. That is, the first element of each pair equals the second. An empty sequence is always complete. A complete sequence identifies the rightmost child of some branch of a tree.

For a list to contain no "gaps":

1. The list must start with a $\chi$ that is initial.
2. If $\chi = a(j, n)b$ and $b$ is complete, then $\chi$ must be followed in the list by $\chi' = a(j + 1, n)c$, where $c$ is initial.
3. The list must end with a $\chi$ that is complete.

Conditions 1 and 3 identify the leftmost and rightmost nodes of the tree. Condition 2 guarantees that every node in the tree has an immediate successor.

In our example, the list obtained is:

$$(1, 2)$$
$$(2, 2)(1, 2)$$
$$(2, 2)(2, 2)$$

Condition 1 is satisfied since (1,2) is initial. Condition 2 is satisfied by $\chi = (1, 2)$ where $a = \lambda$, $b = \lambda$ and $c = (1, 2)$; and by $\chi = (2, 2)(1, 2)$ where $a = (2, 2)$, $b = \lambda$ and $c = \lambda$. Finally, of course, (2,2)(2,2) is complete, so we know that we have all the children of the original symbolic message.

By this technique we may collect all reachable state transitions starting at any procedure $p$ that creates messages of a given type. When all the symbolic messages have returned to their source, the results may be broadcast to other stations.

If desired, $\phi(\alpha)$ may be encoded as a regular expression. This is a consequence of the fact that every finite automaton may be described by a regular expression, and vice versa. The regular expression may be obtained from the state transitions by any standard algorithm, such as appears in [AhHU74]. One regular expression for our example would be:

$$\phi(\alpha) = \alpha\ p_1\ (\ \sigma_1\ (\ p_2 + p_3\ \sigma_3\ p_6\ )\ \sigma_2\ p_4\ )^*\ \sigma_1\ p_3\ \sigma_3\ p_5\ \omega$$

If we are interested only in the procedures (i.e. in $\hat{l}(\alpha)$), then the corresponding regular expression would be:

$$\hat{l}(\alpha) = p_1\ (\ (\ p_2 + p_3\ p_6\ )\ p_4\ )^*\ p_3\ p_5$$

Let us consider the complexity of collecting the state transitions. The symbolic messages traverse each state transition at most once. If there are $k$ message states and $l$ procedures, then there are at most

$$t = l \times k^2$$

state transitions that may be effected by the procedures. The time it takes to traverse these transitions is therefore bounded by $l \times k^2$.

Each state transition is collected at most once by any child of the original symbolic message. The set of all the $\Theta$ therefore takes up at most $O(t)$ space. The $\Sigma$ are similarly bounded since we can only reach new states through the state transitions.

The space taken by the splitting histories is equal to the product of the number of symbolic messages and the average length of each $\chi$. Recall that the splitting histories correspond to a tree with the original symbolic message as its root. Let us assume that the tree is balanced and that each node has the same number of children. If each node has $n$ children (*outdegree n*) and the height of the tree is $h$, then there will be $n^h$ leaf nodes. The space taken by all the paths will therefore be

1. $SPACE = h \times n^h$

The number of edges in the tree is:

2. $n + \cdots n^h$

Each node is the result of at least one state transition, hence:

3. $n + \cdots n^h \leq t$

4. $n^h < t$

5. $SPACE < \log_n(t) \times t$

Since every node in the tree corresponds to at least a split into two children, we know that $n \geq 2$, hence:

6. $SPACE < \log_2(t) \times t$

7. $SPACE < \log_2(l \times k^2) \times l \times k^2$

In the opposite extreme where the tree is severely skewed, we might have a tree where all but the first child of each split dies immediately. Suppose again that each node splits into $n$, and the height of the tree is $h$. Then the space taken by the splitting histories will be:

1. $SPACE = (n-1) + 2 \times (n-1) + \cdots (h-1) \times (n-1) + h \times n$

2. $SPACE < n + 2 \times n + \cdots (h-1) \times n + h \times n$

3. $SPACE < n \times \frac{h \times (h+1)}{2}$

The number of edges in the tree is:

4. $n \times h \leq t$

Again we have $n \geq 2$, so:

5. $h \leq t/2$

6. $SPACE < \frac{t \times \left(\frac{t}{2} + 1\right)}{2}$

7. $SPACE < \frac{t \times (t+2)}{4}$

8. $SPACE < \frac{l \times k^2 \times (l \times k^2 + 2)}{4}$

Of course, these are upper bounds. In general we do not expect procedures to effect transitions between arbitrary states since we would then obtain no useful information about message behaviour. Furthermore, we do not expect every state transition to result in the splitting of a symbolic message, so there should be far fewer nodes in the splitting tree than there are transitions.

### 4.3. Summary

Although we cannot capture message behaviour by exhaustively enumerating all possible message paths, we can still obtain an expression of message flow by partitioning message domains into a finite number of message states. We then extend the notion of a message path to alternating sequences of message states and procedures. The difficulty then lies in determining reasonable partitions of the message domains, and in discovering what the state transitions are.

Since we would like message states to correspond closely to sets of messages that travel together, it is instructive to consider what exactly influences message paths. We develop the idea of a "control attribute" being an attribute whose value somehow influences routing. Routing is directly affected by selection attributes that appear in trigger conditions, and by routing attributes whose values are used to compute new locations for messages. Attributes used in the computation of actions that modify control attributes are also control attributes. A distributed algorithm for determining the control attributes has been presented.

Message states intuitively correspond to predicates over the control attributes. These predicates come from the trigger conditions, and from inverting the actions that modify control attributes. The analysis of state transitions is simplified somewhat by considering message states expressible as conjunctions of simple conditions on individual attributes. Where fields are numeric, message states are products of attribute ranges. The attribute ranges may be obtained at the same time that the control attributes are detected.

State transitions occur when there is a possibility of a message in one state to be mapped to a message in another state, given the presence of the requisite coordinating messages. If message states and actions are reasonably uncomplicated, then the state transitions can be determined without too much difficulty. With very messy states or actions, however, it may be very hard to tell whether a state transition may take place without the help of numerical analysis tools or detailed information on the functional behaviour of the actions. One approach to this problem is to monitor messages as they are handled by a procedure and note what state transitions actually take place. This has the advantage of not requiring any detailed analysis of actions, but has the distinct disadvantage that one never knows for certain that unobserved transitions are impossible.

If an "analysis workstation" is available, then all state transitions can be sent to that station where the information will be processed and analyzed. Alternatively, message paths may be determined by using "symbolic messages" that represent sets of message states that travel from station to station collecting state transitions. This has the advantage of determining only those states and transitions that are actually reachable, and of not putting any undue computational burden on any given workstation. The cost is a greater load on the communications medium to handle these symbolic messages.

# 5. Global behaviour

In this chapter we cover a variety of related topics that fall under the general heading of "global behaviour". We first elaborate on our finite state automata interpretation of message flow, and we show how coordination of messages by procedures can be recovered by deriving a Petri net model that "welds" the automata together. This Petri net model proves useful in later discussions. We also discuss various types of blocking, in which messages are held up indefinitely at a procedure, awaiting some event. The opposite problem, in which procedures are endlessly triggered and messages return repeatedly to the same state, is discussed in the section on procedure loops and message loops. Blocking and looping thus constitute the two extremes of anomolous behaviour in message systems. A section on run-time monitoring rounds off this chapter with some suggestions on alternative ways of detecting procedure loops.

The model of message flow that we have developed is characterized by a collection of message states representing sets of values that messages may acquire, and by expressions of the state transitions that they may undergo. We have not yet explicitly dealt with the matter of coordination between messages of different types, nor have we characterized global behaviour by classifying the ways in which messages may flow through the system.

Some questions about global behaviour are straightforward to handle. One may, for example, easily identify unreachable message states by noting what states are not encountered by any of the symbolic messages in the algorithm described in chapter four. Other questions are not so simple to answer. This has much to do with the fact that we cannot exhaustively enumerate all message paths. Message states sacrifice information by equating message values that may in fact behave rather differently. The better the message states are, the less information that is lost. (A "better" partition may or may not be a finer one.)

We may especially note this loss of information in the non-determinism displayed in the message paths. Although there is a certain amount of non-determinism inherent in the model, spurious "non-determinism" may be introduced by a message state space that is "too coarse". We shall see in an example how this may happen.

Blocking is one aspect of global behaviour. A procedure is blocked if it is waiting for messages that never arrive. There are several reasons why this may happen. Spurious non-determinism is one of the reasons it is difficult to detect some kinds of blocking. Another problem is that since our message paths are only "approximate", we must be careful when we use them to draw inferences about the presence or absence of blocking.

A curious phenomenon that may occur in a partially automated message system is that of message loops. An unfortunate combination of procedures may bat a message back and forth forever, or until someone notices the problem and stops them.

A slightly more unusual problem is the occurrence of "procedure loops". Here we have a set of procedures repeatedly triggering each other through the messages they produce. Although message loops are a special case of procedure loops, we may have a procedure loop with no message loop, especially if the procedures consume their input messages and produce new outputs with each iteration of the loop.

Because our analysis sacrifices some information for the sake of tractability, other techniques that do not make simplifying assumptions about message behaviour can be very useful in detecting some of these problems. We describe, for example, a way of detecting procedure loops at run-time by monitoring the chain of events.

### 5.1. Petri net representation and non-determinism

Before we continue with a discussion of non-determinism, it is instructive to note that there is a natural Petri net interpretation of global behaviour arising from the message states and state transitions derived in the previous chapter.

Although message behaviour can be compared to the behaviour of a finite automaton, this does not tell the whole story since coordination is not explicitly represented. What we in fact have is a *collection* of finite automata, one for each message type, interacting with each other. For procedures to fire, several of these automata must be in the right state at the same time. In fact, it is possible to "weld" these automata together in such a way as to produce a Petri net that captures the procedure interactions. The resulting Petri net not only models the message flow and control flow apparent in the automata, but also captures the coordination of messages by procedures. We thus explicitly represent the flow of messages of all types at once, and the necessary trigger conditions (in terms of message states) of all procedures.

Consider, to begin with, a Petri net with one transition for each procedure, and places for the inputs and outputs of the procedures. Each input and each output may correspond to several message states, however. Let us then add one place for each message state of each message type. Now add transitions from the places representing message states to the places representing inputs whenever messages in those states match the trigger conditions for the procedure. Similarly add transitions from outputs to message states when actions may map messages to those states. In figure 5.1 we represent procedure $p$ with inputs $i_1$ and $i_2$ and outputs $o_1$ and $o_2$ as a single transition. Message states $\sigma_1$ through $\sigma_4$ and $\sigma'_1$ through $\sigma'_5$ are represented by places. Petri net transitions are also present to represent the fact that input $i_1$ corresponds to message states $\sigma_1$ and $\sigma_2$, and that $p$ generates outputs in state $\sigma_4$. An entire Petri net may be built in this way with transitions mapping message states of various types to other message states.

There is a serious problem here, however. In figure 5.1 it appears that messages in states $\sigma'_1$ or $\sigma'_2$ may map to messages in states $\sigma'_3$ or $\sigma'_4$. Suppose that in fact we only have state transitions $p{:}\,\sigma'_1 \mapsto \sigma'_3$ and $p{:}\,\sigma'_2 \mapsto \sigma'_4$. In this case that information would be lost by our Petri net interpretation. It is possible to remedy this situation by adding extra Petri net states to "remember" what the previous message states were. In figure 5.2 we have added states $t_1$, $t_2$, $t'_1$ and $t'_2$ to accomplish precisely that.

We may formalize this construction as follows:

Let $P$ be the set of procedures in the system. $I(p) = < \cdots, I_{pj}, \cdots >$ is the list of input types to $p$. $O(p) = < \cdots, O_{pj}, \cdots >$ is a "copy" of $I(p)$ representing the outputs. $\Sigma_i$ is the set of message states of type $X_i$. $T_i \subseteq \{(p, \sigma_j, \sigma_k) | \sigma_j, \sigma_k \in \Sigma_i,\ p \in \hat{p}(\sigma_j),\ \hat{a}_p(\sigma_j) \cap \sigma_k \neq \emptyset\}$ is the set of state transitions for messages of type $X_i$. There are at most $|P| \times |\Sigma_i|^2$ of these (and, in general, much less). Also, let $r_i = \{(p, \sigma_j) | \exists\ \sigma_k$ such that $(p, \sigma_j, \sigma_k) \in T_i\}$. The $r_i$ represent the $\sigma_j$ that trigger some procedure $p$. We shall use the elements of these sets as labels for the places and transitions of our Petri net.

Let our Petri net have places with labels in:

$$\{I_{pj} | p \in P,\ I_{pj} \text{ in } I(p)\} \cup$$
$$\{O_{pj} | p \in P,\ O_{pj} \text{ in } O(p)\} \cup$$
$$\left(\bigcup_{X_i \in X} \Sigma_i\right) \cup \left(\bigcup_{X_i \in X} r_i\right)$$

and transitions with labels in:

$$P \cup \left(\bigcup_{X_i \in X} r_i\right) \cup \left(\bigcup_{X_i \in X} T_i\right)$$

Note that we have both places and transitions labeled $(p, \sigma_j) \in r_i$, but they are in fact to be considered disjoint. We therefore have places representing message states, procedure inputs and outputs, and "state

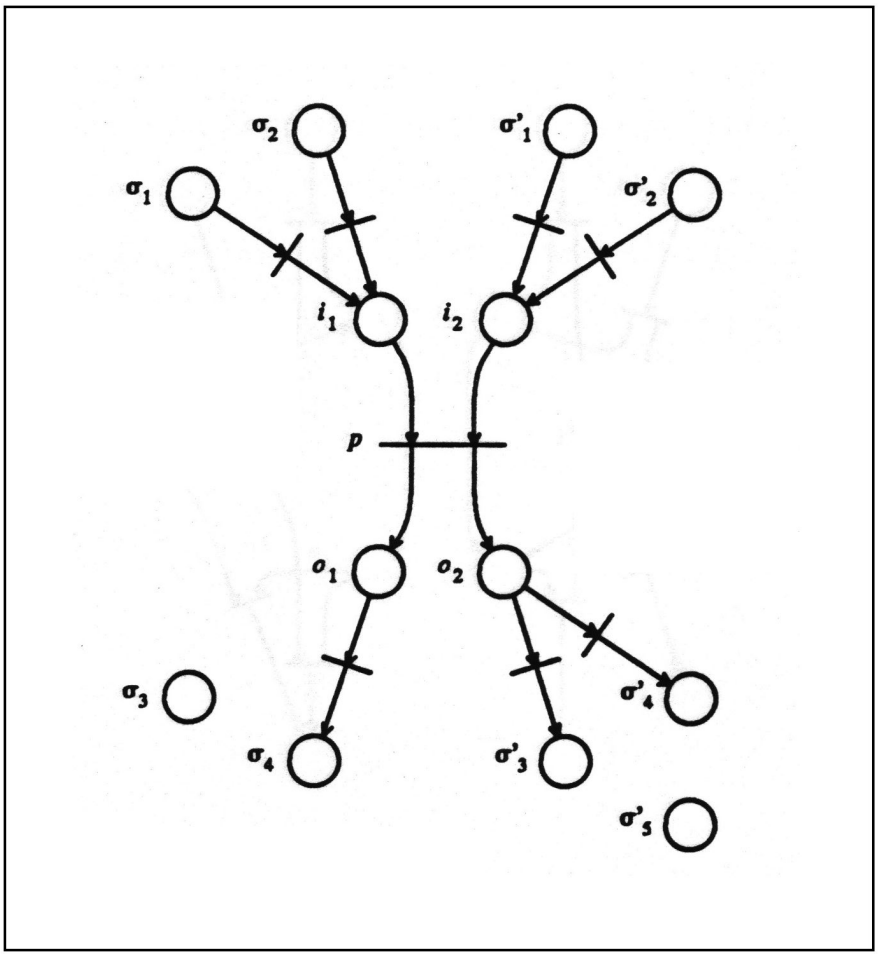



**Figure 5.1** : A Petri net interpretation of message flow

reminders" to remember previous states. The transitions represent procedures and the acts of "grabbing" and "releasing" messages. The "grabbing" and "releasing" allows us to capture the idea that procedure inputs and outputs may correspond to several states.

The transitions have the following inputs and outputs:

1. a transition labeled $p \in P$ has inputs $I(p)$ and outputs $O(p)$,
2. a transition labeled $(p, \sigma_j) \in r_i$ has input $\sigma_j$, and has outputs $(p, \sigma_j)$ and $I_{pk}$ where $I_{pk} = X_i$
3. a transition labeled $(p, \sigma_j, \sigma_k) \in T_i$ has inputs $(p, \sigma_j)$ and $O_{pk}$ where $O_{pk} = X_i$, and has output $\sigma_k$.

It is now clear from the construction that tokens may "travel" from message state $\sigma_j$ to state $\sigma_k$ via procedure $p$ only if there is a state transition labeled $(p, \sigma_j, \sigma_k) \in T_i$. This is the problem that we set out to correct after our first attempt at a Petri net representation. In addition, procedure $p$ may only fire if it has at least one message available for each of its inputs. We have therefore succeeded in "welding" together the finite automata of message flow by reclaiming the coordination that we "sacrificed" in chapter 4.

Note that the Petri net we have obtained is "conservative". (A Petri net is *conservative* if we can assign weights to tokens according to their places so that the net weight of the entire net never changes.) Since tokens represent message instances in certain states, this means that messages are "honestly" represented. We neither gain nor lose messages. To prove this, let us assign double the weight to tokens in the places representing message states. Consider the transition firings in 1, 2 & 3 above. Transitions

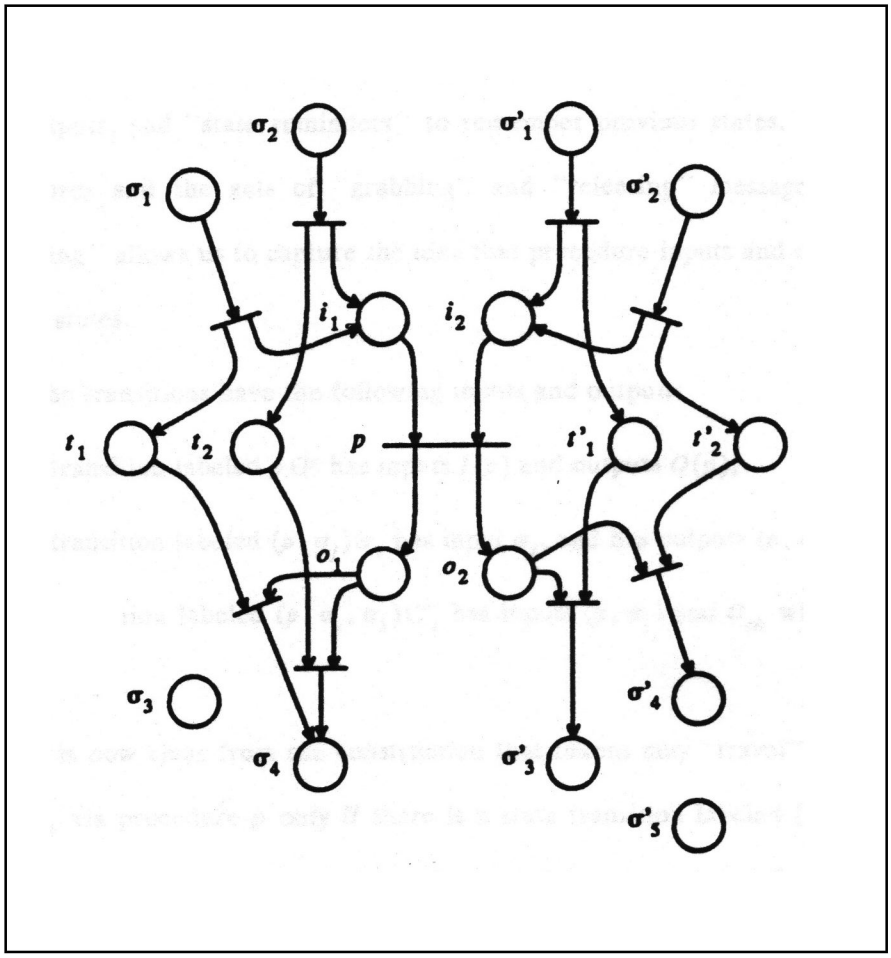



**Figure 5.2** : An "improved" Petri net interpretation

representing procedures are trivially conservative since they all have the same number of inputs as outputs. The "grabbing" and "releasing" transitions are also conservative since the former "splits" a message state token into a procedure input token and a "reminder" token, and the latter "joins" a "reminder" token and a procedure output token. In either case, the total weight of the tokens is the same before and after.

The net is no longer conservative if we add extra transitions to represent the creation and destruction of messages. This may be done by adding one transition for each place representing an $\alpha$ state or an $\omega$ state. Tokens could then be added at will to the $\alpha$ states, and removed from the $\omega$ states. Equivalently, we may simply delete procedure input and output places corresponding to the creation or destruction of messages. Message states $\alpha$ and $\omega$ need not be explicitly represented in this case.

It is important to note the distinction between the Petri nets that we generate from the message flow automata and the Petri nets that appear elsewhere in the literature. In SCOOP [Zism77] and in Taxis [MyBW80, Barr82], Petri nets are used to control office activities. In these systems the Petri nets are explicitly given by the person specifying an office procedure. In our model, however, the Petri net-like behaviour is a side-effect of simple one-step procedures that are only loosely connected. Furthermore, tokens in SCOOP and Taxis represent flow of control rather than flow of message instances. In our model there is a very close relationship between message flow and control flow since messages must be present for procedures to be triggered. Similarly, in Information Control Nets [Elli79, Cook80], which strongly resemble Petri nets in many ways, there is a conscious effort to distinguish between data flow and control flow.

Here, since we are interested in automatically triggered procedures, we make a conscious effort to identify the two.

Finally, the Petri nets that we generate would typically contain far more places (corresponding to message states) than would be required to represent a procedure in SCOOP or Taxis. This is because we wish to translate as much as possible the trigger conditions of a procedure (*productions* in SCOOP) into distinct message states. In this way the Petri net itself rather than any code associated with its transitions more accurately reflects the behaviour of the system in terms of message flow.

Petri nets are non-deterministic models. The non-determinism displayed in our Petri net model of message flow has a variety of sources. Most of these sources are artefacts of our modeling assumptions rather being inherent properties of the systems we are studying. We can identify four main sources of non-determinism, or apparent non-determinism:

1. Coordination: we cannot necessarily predict what coordinating messages will arrive. Since the entire input set must be available for a procedure to be triggered, we cannot predict on the basis of a single message alone what will happen to it. Furthermore, a single message may be capable of simultaneously triggering several procedures at once if the right coordinating messages are present. Transitions may therefore exist from message states to the inputs of several procedures.
2. User input: since this is, strictly speaking, outside of our system, we can at best model it as non-determinism in the actions. Actions can thus be seen as "multi-valued" functions with a particular value being chosen by the user. Procedure outputs can thus be mapped to any of several new message states.
3. Random number generators: applications are conceivable in which a pseudo-random number generator is used in procedure actions. (Such as selecting random lot numbers for testing purposes, or selecting a random message recipient when several will do.)
4. State-space granularity: a message state-space that is "too coarse" will yield a model of message flow that exhibits non-determinism that may not be observable in the real system. This is a consequence of identifying messages that are actually different in significant ways. The following example is used to explain how this may happen.

Suppose we have five procedures handling customer records. $p_0$ is used to create new records. $p_1$ and $p_2$ are fired weekly. $p_1$ automatically increments the *weeks overdue* field. $p_2$ automatically generates messages to Simon if the account is more than 5 weeks overdue. $p_3$ resets the *weeks overdue* field to zero when payment is received. $p_4$ may be used to retire a customer record. The account must not be overdue. There are three states of interest. $\sigma_0$, $\sigma_1$ and $\sigma_2$ represent records with an *overdue* field of 0 weeks, 1 to 5 weeks, and over 5 weeks, respectively. We have the following state transitions:

$$
\begin{aligned}
&p_0\colon \alpha \rightarrow \sigma_0\\
&p_1\colon \sigma_0 \rightarrow \sigma_1\\
&p_1\colon \sigma_1 \rightarrow \{\sigma_1, \sigma_2\}\\
&p_1\colon \sigma_2 \rightarrow \sigma_2\\
&p_2\colon \sigma_2 \rightarrow \sigma_2\\
&p_3\colon \sigma_0 \rightarrow \sigma_0\\
&p_3\colon \sigma_1 \rightarrow \sigma_0\\
&p_3\colon \sigma_2 \rightarrow \sigma_0\\
&p_4\colon \sigma_0 \rightarrow \omega
\end{aligned}
$$

The next state function $\delta$ would map $(\sigma_1,\ p_3)$ to $\sigma_0$, for example. We represent this graphically in the (non-deterministic) finite automaton of figure 5.3.

A regular expression representing the possible message paths is:

$$\phi(\alpha) = \alpha\ p_0\ \sigma_0\ (p_3\ \sigma_0 + p_1\ \sigma_1\ (p_1\ \sigma_1)^*\ (p_3\ \sigma_0 + p_1\ \sigma_2\ (p_1\ \sigma_2 + p_2\ \sigma_2)^*\ p_3\ \sigma_0))^*\ p_4\ \omega$$

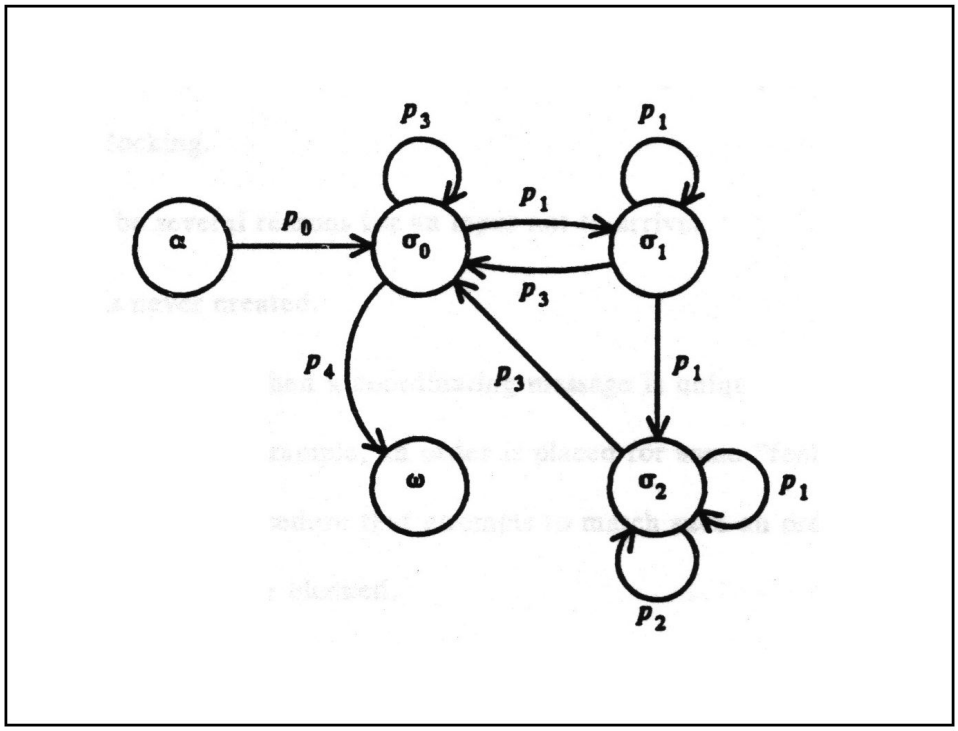



**Figure 5.3** : Non-determinism in message flow

In this example it appears that $p_1$ may fire indefinitely without ever causing the message to change from state $\sigma_1$ to state $\sigma_2$. That is, in state $\sigma_1$, the effect of firing $p_1$ is not uniquely determined. This is not so, of course, since after firing five times, the message will reach state $\sigma_2$. The apparent non-determinism could be removed by creating separate state for messages whose *weeks overdue* fields have the values 1, 2, 3, 4 and 5.

Although it is easy to point this out in an example such as this one, it is quite another matter to do it automatically. If, for example, we try to subdivide $\sigma_2$ then we end up with an infinite number of states, which will never do. (This problem was discussed in chapter four at the end of the section on obtaining message states.) This is one of the costs of sacrificing information in order to make the problem of characterizing message flow tractable.

Although non-determinism is a characteristic of these systems (specifically, when messages may trigger multiple procedures), our means of modeling message behaviour may introduce non-determinism that is not observable in reality. This apparent non-determinism may sometimes be eliminated by fine-tuning the state-space. One must therefore exercise care in drawing conclusions about message behaviour where non-determinism is concerned.

### 5.2. Blocking

A procedure is *blocked* if it waits indefinitely for one of its inputs to arrive. If the procedure has only one input, then that simply means the procedure does not fire, but there may not necessarily be any far-reaching effects. If, on the other hand, the procedure does have other inputs, then inputs that arrive to be processed by that procedure may wait forever because of the blocking.

There may be several reasons for an input not to arrive:

1. The input is never created.

This causes blocking when a coordinating message is uniquely determined, but does not, in fact, exist. If, for example, an order is placed for some "feeblevetzers", and no such items exist, then a procedure that attempts to match such an order with a corresponding inventory record will be blocked.

2. The message states corresponding to the trigger conditions of the procedure are unreachable.

   This may happen because the message reaches a dead end, or because it enters an infinite loop, or it may simply be that all possible paths avoid the procedure in question.

3. The message states corresponding to the trigger conditions of the procedure are avoidable.

   Messages of the input type in question may be able to reach the procedure to trigger it, but alternative paths may avoid it entirely. Blocking may occur here if the message is uniquely determined by the other inputs. An order form that is to be matched against an inventory record for "veeblefetzers" will be unable to proceed if the inventory record happens to be routed along a path that avoids it. (We assume that there is a unique inventory record for any given item.) If, on the other hand, an inventory record is waiting to be matched against an order form, then it may not matter that the order form can be routed along alternative paths -- there will be other orders for that item, so the procedure will not necessarily be blocked.

4. There is a "blocking loop".

   Two procedures are each waiting for a message that is stuck at the other. This is what is most commonly thought of when we speak of "deadlock" in systems where there is contention for resources. The resources in our case are the messages.

5. The missing input is itself stuck at another procedure that is blocked.

   The other procedure may be blocked for any of the first four reasons.

Note that in cases 1, 3, 4 and 5 we only have blocking if the awaited message is uniquely determined by the other inputs. If it is not, then another message in the same state may eventually arrive, so we would not have blocking. For example, since order forms would not be uniquely determined by any procedure matching them against inventory forms, they could never be the cause of blocking in such a situation. In case 2, we have blocking even if the awaited message is not uniquely determined since *no* message may ever reach the desired state.

Let us consider each of the cases in turn.

### 5.2.1. Message creation

The first case seems a degenerate one, and not so much a candidate for analysis. At any rate, one may easily identify all the procedures that are responsible for creating messages of the awaited type. Possibly this information can be useful in determining whether the awaited message has been created. If we can determine that procedure $p$ may not be supplied with some inputs for this reason, we say that $p$ is *1-blocked*, or *1-BL*, for short.

Of course, if the procedure creating the messages is blocked, then no messages will be created. This may be considered an instance of case 5, however.

### 5.2.2. Unreachable states

Cases 2 and 3 are quite similar in that we are interested specifically in the message paths. In case 2 it is simply a matter of determining whether the message states corresponding to the trigger condition of a procedure are reachable or not. This information is readily available from our work in chapter 4. The symbolic messages only encounter reachable message states. Lists of reachable and unreachable states can thus be compiled.

Exactly *why* a particular message state is not reachable is another matter. A characterization of message flow may be useful in tracking down what is wrong, but it is well-nigh impossible to tell this without a deeper understanding of what the procedures are supposed to do. There are, however, two readily identifiable situations that suggest that something is amiss:

i) A message may hit a *dead end*.

A message that ends up at a location where no procedure is prepared to handle it at all is at a "dead end". Without user intervention the message will stay there forever. A dead end may be the consequence of incorrect routing. Naturally this will prevent a message from reaching waiting procedures. Again, we may discover dead ends by our analysis of chapter 4. Symbolic messages terminate when a message is destroyed, or it reaches a state that has been seen by some other symbolic message, or it reaches a state that is a dead end.

ii) A message may enter an infinite loop.

This happens if a message reaches a set of mutually reachable states from which there is no escape. States *outside* that set would not be reachable. In particular, $\omega$ could never be reached. This too may be the result of incorrect routing. In a directed graph, a set of mutually reachable nodes is called a *dicomponent* [BoMu76], or a *strongly connected component* [AhHU74]. Once a message leaves a dicomponent it may (by definition) never return. If the dicomponent cannot be left, then the message is in an infinite loop. A depth-first search algorithm can partition a directed graph into its dicomponents in order $O(\max(n, e))$, where $n$ is the number of nodes and $e$ is the number of edges [AhHU74]. To identify infinite loops, one need only determine whether there are any dicomponents with no arcs leaving it for another dicomponent.

A procedure for which some input cannot arrive because the input message states are not reachable is *2-blocked*, or *2-BL*.

### 5.2.3. Avoidable states

In case 3 we are concerned with messages that may or may not arrive. A state may be reachable, but not necessarily by all messages of the specified type. Blocking is possible if any given message is not guaranteed to reach at least one of the message states corresponding to the trigger condition, *and* that message is uniquely determined by one of the other inputs. To determine the latter, one needs to know something more about constraints on the messages. If, for example, we know that a certain field of a message is a key field, and we have a procedure that matches that message against another via that key field, then we know that for any matching input it is uniquely determined. An inventory record, for example, is uniquely determined by any order form.

As to the matter of reachability, we may rephrase it as follows: Is it possible for messages of a given type to avoid *all* of the message states corresponding to the trigger condition for a given procedure? In figure 5.2, message states $\sigma_1$ *and* $\sigma_2$ must be simultaneously avoidable for input $i_1$ to be avoidable. In this light it is clear that we may easily answer this question. One need simply traverse the directed graph of the message state automata, starting at $\alpha$, and avoiding all nodes representing message states that are inputs to that procedure. If we can construct a path to $\omega$ that avoids all these nodes, then it is possible for a message never to trigger the procedure in question. Clearly we need only traverse each edge of the graph at most once, so the problem is solvable in order $O(t)$, where $t$ is the number of state transitions (i.e. the number of edges in the graph). If all paths encounter at least one of the input states, then they are unavoidable (as a set), and this cannot be a source of blocking.

If the reachable message states corresponding to some input of procedure $p$ are all avoidable, then $p$ is *3-blocked*, or *3-BL*.

### 5.2.4. Deadlock

There is the possibility of deadlock, wherein two procedures are each waiting for a message held by the other.

Suppose that procedure $p$ has some input $x$ that uniquely determines some other input $y$. Suppose also that $y$ may come to $p$ from $p'$, and it uniquely determines some input $z$ at $p'$. Finally suppose that $z$ comes to $p'$ from $p''$, where $z$ uniquely determines the same $x$ of procedure $p$. We then have a potential deadlock in which $x$ waits at $p$ for $y$, $y$ waits for $z$ at $p'$, and $z$ waits for $x$ at $p''$.

Let us suppose that we know for all procedures $p$ when some input $X_i \in I(p)$ uniquely determines some other input $X_j \in I(p)$, *and* there is no other procedure $p'$ accepting messages of type $X_i$ in the same states as those accepted by $p$. Messages of type $X_i$ must therefore wait at $p$ for the arrival of some *specific* message of type $X_j$. A message of type $X_i$ would uniquely determine one of type $X_j$ whenever we have some trigger condition of the form $x_n = y_m$ where $x \in dom(X_i)$, $y \in dom(X_j)$ and $X_{jm}$ is a key field of messages of type $X_j$. We represent this information as a set of tuples:

$$AWAITS \subseteq \{(p, X_i, X_j) | p \in P, X_i, X_j \in X\}$$

For $(p, X_i, X_j) \in AWAITS$, we say that $p: X_i \rightarrow X_j$, or simply $X_i \rightarrow X_j$. Furthermore, we say that:

$$X_i \overset{*}{\rightarrow} X_k$$

if we have a sequence:

$$X_i \rightarrow X_j \rightarrow \cdots \rightarrow X_k$$

If $p: X_i \rightarrow X_j$, then messages of type $X_i$ must *await* uniquely determined messages of type $X_j$. Similarly, if $X_i \overset{*}{\rightarrow} X_k$, then messages of type $X_i$ must await messages of type $X_k$, since the latter are uniquely determined by the former.

If $X_i \overset{*}{\rightarrow} X_j$, *and* $X_j \overset{*}{\rightarrow} X_i$, (i.e. $X_i \overset{*}{\rightarrow} X_i$) then a message of type $X_i$ awaits a message of type $X_j$ and vice versa. If the "two" messages of type $X_i$ are in fact one and the same, then we have the distinct possibility of deadlock. We need only find ourselves in the situation where messages of type $X_i$ and $X_j$ are awaiting each other at precisely the same time. Since there is no other procedure that these messages can trigger, then they will both wait forever, neither able to reach the other.

The set *AWAITS* of dependencies defines a directed graph with nodes in $X$ and arcs in *AWAITS*. $X_i \overset{*}{\rightarrow} X_i$ occurs precisely when there is a cycle in the directed graph. Cycles, of course, occur within the dicomponents of the graph. As we mentioned earlier in this section, dicomponents can easily be determined by a standard algorithm such as in [AhHU74]. Any dicomponent with more than one node in it would yield an instance of $X_i \overset{*}{\rightarrow} X_j$, and would therefore provide us with a potential deadlock.

If a procedure $p$ can be blocked due to deadlock, then we say that $p$ is *4-blocked* or *4-BL*.

### 5.2.5. Recursive blocking

Finally, blocking in one procedure may cause blocking in other procedures. If the first procedure is preventing messages from moving on, then other procedures waiting for those messages will also be blocked.

To detect recursive blocking we must find out not only which states are unreachable or avoidable, but also which states are "blocking states". We call a message state a *blocking state* (*BL-state*) if every procedure effecting a transition to that state is blocked, that is:

$$\text{for each } (p, \sigma, \sigma') \in T_i,\ p \text{ is blocked } \Leftrightarrow\ \sigma' \text{ is a blocking state}$$

Conversely, if every state leading to an input of some procedure $p$ is a blocking state or is unreachable, then that procedure is *5-blocked*, or *5-BL*. This is a consequence of the fact that blocking states are a variation on unreachable states -- they are unreachable only as a result of other blocking.

Similarly, if an input is uniquely determined, and the reachable, non-blocking states are all avoidable, then the procedure is *6-blocked, or 6-BL*. We therefore end up with a recursive form of blocking.

We may summarize potential blocking detection in the following algorithm to be run at all stations ("new" BL-states mentioned in step 8 come from steps 7 or 13, whichever is appropriate):

1. **for each** procedure $p$ **do** {
2. **for each** input $X_j \in I(p)$ **do** {
3. **if** $p$: $X_i \rightarrow X_j$ **then**
4. check if $p$ is 3-BL
5. **else** check if $p$ is 2-BL }
}
6. determine which $p$ are 4-BL
7. identify all BL-states arising from the above
8. **for each** $p$ not BL, such that $(p, \sigma, \sigma') \in T_i$ where $\sigma$ is a new BL-state **do** {
9. **for each** input $X_j \in I(p)$ **do** {
10. **if** $p$: $X_i \rightarrow X_j$ **then**
11. check if $p$ is 6-BL
12. **else** check if $p$ is 5-BL
}
}
13. identify all new BL-states arising from the new 5-BL or 6-BL procedures, if any
14. **if** there are no new BL states **then** STOP
15. **else** continue from step 8

Steps 4, 5, 6 and 7 are as described earlier in this section. Steps 11 and 12 are similar to 4 and 5.

The algorithm must terminate since there are only a finite number of procedures and a finite number of states. As long as the algorithm continues to run, at least one new BL-state must be found at step 13. Eventually we must run out of candidates for BL-states. Similarly, we eventually run out of candidates for 5-BL or 6-BL procedures.

The blocking that we uncover can be of interest in several ways. If a procedure $p$ is 2-BL, then we know that it cannot fire under normal circumstances. This means that (according to our analysis) there is at least one input to the procedure for which there is no known path to the procedure. This may mean that $p$ is incorrect, in the sense that it has been created under the delusion that its inputs *will* arrive, or it may mean that some incorrect procedure elsewhere is improperly routing messages, possibly to dead ends, or into message loops. An examination of the message flow automaton will reveal how it is being routed, and possibly provide some insight into what the problem is.

If procedure $p$ is 3-BL, then that means that a uniquely-determined input is (theoretically) capable of avoiding $p$. An examination of the path that does (appear to) avoid $p$ can provide insight into whether there is truly a problem or not. Note that our analysis may have generated spurious paths, if there are state transitions present in our model that for some reason never take place in the running system.

Procedure $p$ and $p'$ are 4-BL if there is some theoretically possible configuration in which $p$ and $p'$ are each preventing the progress of messages required by the other procedure. It remains for someone to look more closely at that configuration to tell whether it is in fact reachable in the running system. If it is, then we can either modify the procedures to avoid the blocking, or we can monitor the flow of these messages to detect blocking if it ever occurs.

Procedures that are 5-BL or 6-BL are only blocked if message inputs are stuck at a blocked procedure. Naturally, if we solve the blocking at the other procedure, or if that blocking is not reflected in the running system, then the 5-BL or 6-BL problem goes away.

### 5.3. Procedure loops

Infinite loops may be thought of as the opposite extreme to blocking and deadlock. In the case of blocking we had problems with messages being "stuck" and nothing happening as a consequence. Here we have problems with too much happening. Messages either loop endlessly, visiting the same stations and procedures, or procedures are fired repeatedly, creating an undending stream of messages. We shall discuss here the kind of infinite loops that may arise, and how we may go about detecting them. The different kinds of loops all turn out to be variations on what we call "procedure loops". Our Petri net model provides us with an analytical approach to detecting when procedure loops may occur.

Our discussion of message loops earlier revealed that there may be situations in which messages encounter the same states infinitely often. This may happen naturally with certain messages that are in fact records expected to be handled repeatedly and indefinitely in more-or-less the same way. The inventory records of a previous example are repeatedly processed by the same procedures whenever new order forms arrive. This sort of message loop does not cause any problems since the inventory records must wait before they are processed again. If, on the other hand, they do not have to wait, then we may have a message loop that is unmoderated. Procedures will fire repeatedly, as fast as they possibly can until someone notices the problem and repairs it.

Unmoderated message loops can be thought of as a special case of *procedure loops*. A procedure loop exists when a given configuration of procedures and message instances provides the opportunity for some procedures to fire infinitely often without human intervention. Every unmoderated message loop, then, is clearly part of a procedure loop. Some procedure loops, however, may not contain any message loop. Consider figure 5.4. Procedure $p$ generates message $x$, which is consumed by procedure $p'$. $p'$ in turn generates $y$, which triggers $p$. We have a procedure loop, but no message loop exists since all messages handled by $p$ and $p'$ have finite paths.

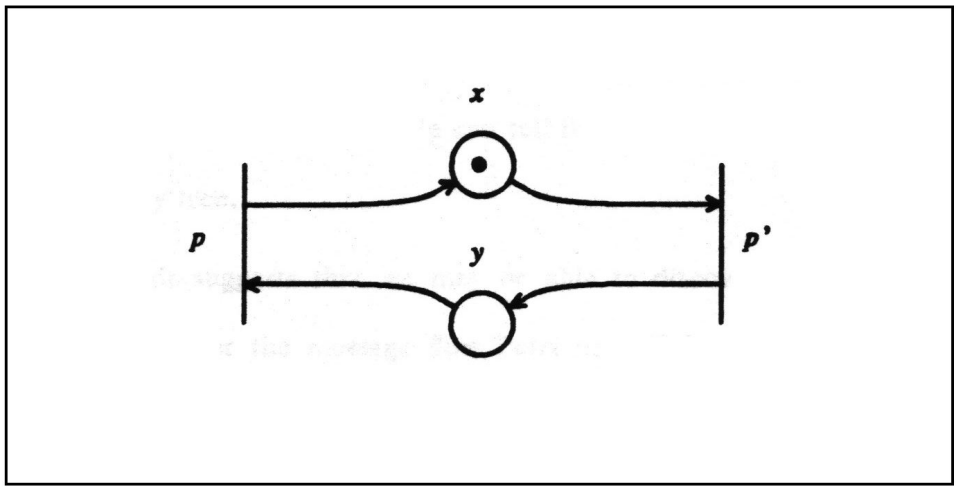



**Figure 5.4** : A procedure loop

Procedure loops depend not only on the presence of an unusual configuration of procedures, but also on a corresponding configuration of messages to start the "chain-reaction". Our Petri net interpretation of message flow can help us now. A Petri net can represent the interaction of procedures (up to the accuracy of the message state-space partition), and a marking of that net can represent the current message states of all the messages in the system. We limit our Petri net to those procedures that do not require any user input. A procedure loop exists if the Petri net can be fired forever. This may happen if and only if there is some transition firing sequence that may be repeated infinitely often [KaMi69]. Such a sequence must yield a new marking that is "at least as big as" the initial marking, that is, the sequence must at least restore all of the tokens used. If $\mu$ is a marking of the Petri net, and $t_1 \cdots t_n$ is a transition firing sequence yielding new marking $\mu'$, then $t_1 \cdots t_n$ can be repeated infinitely often if $\mu_i \leq \mu'_i$ for each $i$.

Karp and Miller in [KaMi69] describe a *reachability tree* which summarizes the markings reachable from an initial marking $\mu$. The nodes of the tree are markings and "pseudo-markings". We draw an arc between two nodes if there is some transition that is enabled in the first marking, and results in the second marking when it is fired. Whenever a marking is reached that is strictly greater than some previous marking, then the greater components are replaced by the symbol $\omega$ (not to be confused with the $\omega$ of message flow model). Markings containing an $\omega$ are called *pseudo-markings*. Any node that is identical to any of its ancestors is made a terminal node. This guarantees that the reachability tree be finite (see [Pete83] for a readable proof of this result). We can tell if the Petri net can be fired forever by examining the reachability

tree.

This result suggests that we may be able to discover procedure loops by generating a reachability tree for the message flow Petri net. This may, in fact, be done for any given marking of the Petri net (i.e. any given configuration of messages in the system). Our attempt is frustrated, however, by the fact that we wish to determine whether procedure loops may exist for *all* possible markings. The reachability tree answers the question: For a given marking, can this Petri net be fired forever? What we would like to know, however, is: Does this Petri net have a marking from which it may be fired forever?

An obvious approach is to try to solve the problem *symbolically*. We may generate a reachability tree starting with an initial marking consisting of variables rather than constants. If we ever reach a marking that is at least as big as the initial marking, then we are done. No matter that we do not have specific values for the initial marking since every reachable marking can simply be computed relative to the initial one. Unfortunately this plan has a severe problem. Crucial to the algorithm for generating the reachability tree is the criterion that the tree be finite. This is done by showing that any infinite sequence of transitions must eventually yield a marking that is at least as big as some ancestor. When we start with a symbolic initial marking, the argument no longer holds. Since the initial marking may be arbitrarily large (though finite), different components may grow and shrink without bound.

Although there may be a way to "patch" the reachability tree for symbolic markings, there is another approach that easily yields a solution. Petri nets are equivalent to *vector addition systems* [KaMi69]. This alternative representation encodes the transitions of a Petri net by using two matrices, $A^-$ and $A^+$. Each matrix has $n$ rows and $m$ columns, where $n$ and $m$ are the number of places and transitions, respectively. The $(i, j)$ entry of $A^-$ is $-1$ if place $i$ is an input to transition $t_j$ and the $(i, j)$ entry of $A^+$ is $+1$ if place $i$ is an output to transition $t_j$. For the net in figure 5.4, we have:

$$A^- = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \text{ and } A^+ = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

with $p$ and $p'$ represented by the first and second columns of each matrix, respectively.

Transition $t_j$ is enabled in marking $\mu$ if $\mu + A^-{}_j \geq 0$ (where $A^-{}_j$ is the $j$th column of $A^-$). Suppose $A = A^- + A^+$. In our example:

$$A = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

If $t_j$ is enabled in $\mu$, then the result of firing $t_j$ is $\mu' = \mu + A_i$. Furthermore, if we have a sequence of transitions that can be fired from $\mu$, and we represent that sequence by a column vector $x$ where $x_j$ is the number of times $t_j$ is fired, then $\mu' = \mu + Ax$ is the marking that results after firing the sequence.

If we can find some non-negative integer column vector $x \neq 0$ such that $Ax \geq 0$, then $\mu' = \mu + Ax > \mu$, so that any transition sequence represented by $x$ can be fired indefinitely, starting from some appropriate initial marking $\mu$. Furthermore, we can always find a marking $\mu$ "big enough" that the transition sequence represented by $x$ can be fired at least once. The marking $\mu = -A^- x$, for example, guarantees this. Consequently, we have a procedure loop if and only if there is some $x$ such that $Ax \geq 0$. The question only remains whether we can easily solve $Ax \geq 0$. To this end we present the following theorem:

**Theorem 5.1** : The problem, "Does a Petri net have a marking in which some transition sequence can be fired infinitely often?" can be solved in polynomial time.

**Proof** : By reduction to linear programming. Let $A$ be the matrix encoding the transitions of the Petri net, as described above. Then the problem is solved if we can answer whether there exists a non-negative integer column vector $x \neq 0$ such that $Ax \geq 0$. Let $A'$ be the matrix obtained by adding a column of zeroes at the left side of $A$, followed by a row of ones at the top of $A$. $A'$ is therefore an $(n+1) \times (m+1)$ matrix such that:

$$A'_{ij} = \begin{cases} A_{ij} & \text{if } i \geq 1,\ j \geq 1 \\ 0 & \text{if } i \geq 1,\ j = 1 \\ 1 & \text{if } i = 0 \end{cases}$$

Intuitively this corresponds to adding one place, $p_0$, which is an output of every transition, and adding one transition, $t_0$, whose only output is $p_0$. Consequently, $p_0$ serves to *count* the total number of transition firings.

Consider the linear programming problem $A'\ x' \geq (1, 0, \ldots, 0)^T$ where we seek to minimize the cost function $cx'$, $c = (1, 0, \ldots, 0)$. (If $v$ is a row-vector, then $v^T$ is the column-vector, *v transpose*.) The cost is therefore $x'_0$, the number of times that we need to fire $t_0$.

The constraint $A'\ x' \geq (1, 0, \ldots, 0)^T$ guarantees that at least one transition fire, since each transition places a token in $p_0$. Furthermore, $x' = (1, 0, \ldots, 0)^T$ is a basic feasible solution, since transition $t_0$ places a token in $p_0$. The cost of this solution is 1, since $t_0$ fires once. This is therefore an upper bound on the cost. The lower bound is 0, corresponding to a solution $x'$ that does not use $t_0$. Such a solution would also be a solution to our original problem, since it guarantees that we fire only transitions represented by $A$.

Furthermore, the solution is always either zero or one. Suppose that we have a solution such that $cx' = x'_0$ lies between 0 and 1. (Such a solution would correspond to a "fractional" number of firings of $t_0$.) Consider $x' = x'' + x'''$ where:

$$x''_i = \begin{cases} 0 & \text{if } i = 0 \\ x'_i & \text{if } i \neq 0 \end{cases} \text{ and } x'''_i = \begin{cases} x'_0 & \text{if } i = 0 \\ 0 & \text{if } i \neq 0 \end{cases}$$

Now

$$\begin{aligned} A'\ x'' + A'\ x''' &\geq (1, 0, \ldots, 0)^T \\ A'\ x'' &\geq (1, 0, \ldots, 0)^T - A'\ x''' \\ A'\ x'' &\geq (1 - x'_0, 0, \ldots, 0)^T \end{aligned}$$

Since $(1 - x'_0) > 0$, there exists some $k$ such that $k(1 - x'_0) > 1$, so

$$A'\ k\ x'' \geq (1, 0, \ldots, 0)$$

but then $c\ kx'' = 0$, a contradiction to our assumption that the minimum lay between 0 and 1.

The linear programming problem has a solution with cost 0 if and only if $Ax \geq 0$ has a solution $x \neq 0$. This is easily seen by letting $x_i = x'_i$ for all $i > 0$. Furthermore, $x'$ cannot be all zero else $A'\ x' = 0$, violating our constraint, $A'\ x' \geq (1, 0, \ldots, 0)^T$. Hence $x$ is a non-zero solution. Finally, $x'$ may be non-integral, but linear programming always yields rational solutions. Since $x'$ is a rational solution, there exists a positive integer $k$ such that $kx'$ is integer. Furthermore, if $x'$ is a solution, then clearly so is $kx'$. This then yields an integer solution for $x$, if one exists.

Since linear programming is solvable in polynomial time in the size of the input (by the ellipsoid method [PaSt82]), so is infinite fireability of Petri nets. □

Since there is a polynomial-time solution to the procedure loop problem, one is left wondering if there is not some way of making the reachability tree approach work. This would be especially desirable since the linear programming solution destroys the intuition behind the problem: there is no notion of "fractional procedure firings", though linear programming yields rational solutions. Note we do not have to resort to integer-linear programming, since we are not really interested in minimizing anything. The cost function that we construct is there mainly as a tool to count transition firings. In the end we merely wish to find *any* solution, not some "optimal" one, in any sense. Since a rational solution yields an integer one, we may exploit the polynomial nature of regular linear programming.

## 5.4. Run-time monitoring

We have mentioned several times that our analysis suffers from our inability to exhaustively enumerate message paths. Message states inevitably hide some of the differences between "similar" messages. We would therefore like to develop some techniques that supplement our analysis by monitoring system behaviour at run-time.

Some problems are trivial to identify: dead ends are easily recognized if a message fails to satisfy trigger conditions of any of the procedures it is currently exposed to. Similarly we may take note of nondeterminism whenever we find a message triggering two procedures simultaneously. (Of course, to do so we would need to list all the procedures the message may trigger *before* firing any of them; afterwards it may be too late.)

Less trivial to detect are message loops and procedure loops. This is of special concern since they are examples of the system "gone wild". At best these loops needlessly consume cpu cycles. At worst they may saturate the file system and the communications medium with endlessly generated messages. If we can detect these loops when they occur, then we can arrest the damage they may inflict.

In the following two sections we suggest ways of tackling these problems at run-time. Unfortunately (as we shall prove), there are no effective procedures for conclusively detecting message loops or procedure loops, but we can nevertheless apply some techniques that will tell us where potential loops exist.

### 5.4.1. Message loops

Message loops are generally of concern if they are unmoderated, that is, if a message in the loop can repeatedly trigger procedures without ever having to wait for user input or a coordinating message. Message loops are therefore potentially detectable by keeping track for every given message after it has been processed by a procedure whether it can immediately trigger another procedure or not. We call a sequence of procedures that a message triggers without any waiting an *unmoderated triggering sequence*. If such a sequence terminates (with the message waiting, or being destroyed) then the message is not in an unmoderated loop.

In figure 5.5 we see that message $x$ is in an unmoderated loop. Its unmoderated triggering sequence is $p\ p'\ p \cdots$. Here, $y$ and $z$ are in loops, but each must wait for $x$ or $x'$ before they can trigger $p$ or $p'$, respectively.

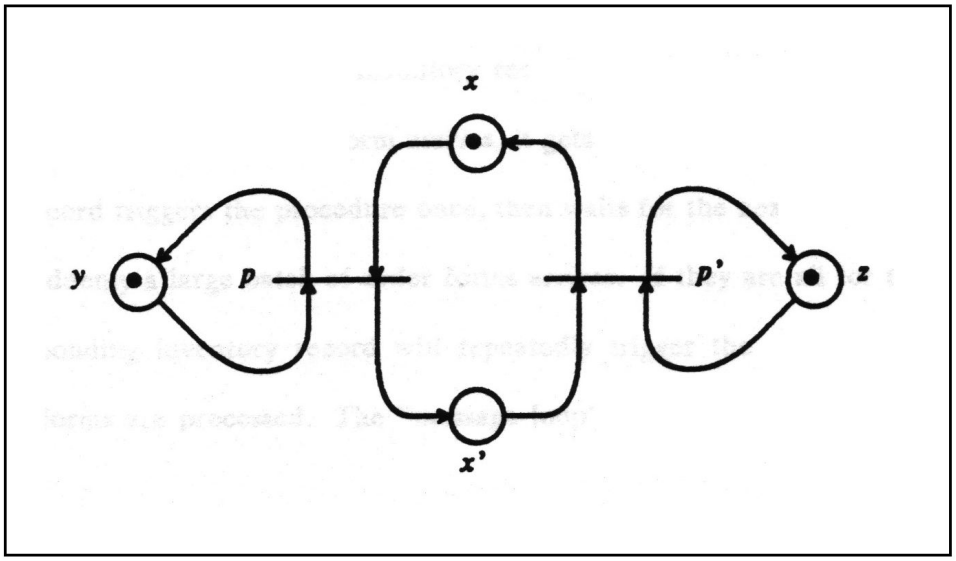



**Figure 5.5** : An unmoderated message loop

How long must an unmoderated triggering sequence get before we are guaranteed of having a loop? Unfortunately this is the halting problem: we may model a Turing machine by using a list attribute of some message to represent the (infinite) tape of the Turing machine, and a procedure to model the logic of the Turing machine (since our actions are completely general). The trigger condition of the procedure would be that the message not represent a halting state of the machine. The action would be to compute the next state. If we could effectively tell whether messages are in a loop or not we would have a solution to the halting problem. This problem is known to be undecidable [HoUl79].

The only reason that we can "detect" message loops and procedure loops is that we approximate system behaviour with a simpler model. In fact, the procedure loops and message loops that we discover in earlier sections may not (as we have pointed out) necessarily correspond to real loops.

When we attempt to detect loops at run-time we are faced with the same problem. If a message does "stop" and wait when an unmoderated triggering sequence terminates, we know that we do not have a loop. If, on the other hand, it does not stop, then we have no effective way of determining whether it will ever stop or not.

At best, we may apply some rules that help us tell if something does appear to be going awry. Clearly, a message loop requires that some message encounter at least one procedure infinitely often. Furthermore, we can safely assume that most unmoderated triggering sequences will be quite short. If an unmoderated triggering sequence for a given message instance does reveal that that message has encountered some procedure twice, then we will probably want to sound the alarm.

Immediately, an exception comes to mind: Consider our old example of a procedure that coordinates order forms with inventory records. The inventory records don’t normally leave the station. When an order form arrives, it gets processed, and leaves the station. The inventory record triggers the procedure once, then waits for the next order form to arrive. Suppose that suddenly a large batch of order forms arrives. If they are all for the same item, then the corresponding inventory record will repeatedly trigger the same procedure until all of the order forms are processed. The "message loop" terminates normally when the forms are all gone.

This suggests that we may wish to keep track of the identities of the coordinating messages at each procedure that we visit. If we visit the same procedure twice in an unmoderated triggering sequence, and we see precisely the same messages as the input, then we may suspect an infinite loop. Although this means that we would not catch the spurious loop of the example above, we would also miss more unusual loops within which our coordinating messages are consumed and created.

It appears, therefore, that if we try to be too clever about our attempts to catch certain potential message loops and not others, then we may end up outsmarting ourselves and miss ones that we should be catching. This is not unexpected, since we know there can be no effective procedure to detect exactly the infinite loops. Better, then, that we trap all unmoderated triggering sequences in which some procedure is encountered twice (or some bounded number of times), and let the user decide whether there is a real problem or not. In cases where we are confident that there is no danger of an infinite loop, we may disable the trap for that trigger sequence, or increase the bound on the number of iterations before we are notified.

### 5.4.2. Procedure loops and daemons

We have already identified message loops as special cases of *procedure loops*. Consequently we know that the detection of procedure loops is also undecidable. We can at best hope to find some means of detecting potential procedure loops. The apparent loops that we find may or may not correspond to true infinite loops. By detecting the start of a potential loop, however, we may be able to prevent the damage done by real loops.

At this point we should note that it is possible for a "moderated" message loop to exist that is part of a procedure loop, *without* any messages being created or consumed within the loop. This means that our technique of tracing unmoderated triggering sequences in order to detect message loops may miss certain esoteric loops. Consider the Petri net in figure 5.6.

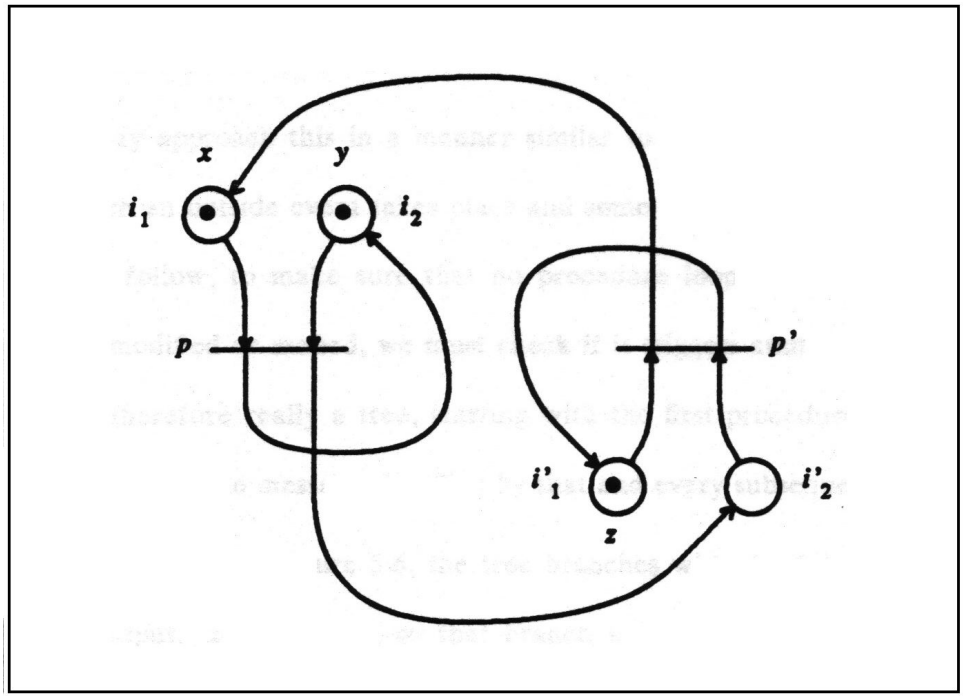



**Figure 5.6** : "Moderated" message loops

Messages $x$, $y$ and $z$ are clearly in message loops, since we can repeatedly fire $p$ and $p'$. We therefore have an unmoderated procedure loop (assume no user input or time conditions at $p$ or $p'$). Note, however, that after firing $p$, message $x$ must wait at $i_2$, since there will be no message at $i_1$. $p'$ then fires, and $y$ is moved from $i'_2$ to $i'_1$, and $z$ from $i'_1$ to $i_1$. $y$ must then wait at $p'$. $z$ then triggers $p$ and then *it* must wait.

Procedures $p$ and $p'$ both grab a pair of messages, and send the one that's been waiting. Since every message must eventually wait, all unmoderated triggering sequences trivially terminate with length one. This means that it is not enough to look at message loops alone. We must consider how procedures interact by looking at all of the messages involved.

We may approach the problem by noting that procedures are event-driven. If the system is in a "quiet" state, that is, no procedure is executing and none are enabled, then the only possible way for a procedure to be triggered is if some outside event occurs, namely some time condition is met, or some user action takes place. We consequently only need to check the trigger conditions of procedures that may be affected by the events that take place. A "chain of events" then occurs, with the actions of procedures causing new events (creating, modifying, mailing and destroying messages), until the system is quiet again. If a "chain of events" is circular, however, then the system will never become quiet, and we have a procedure loop.

We may approach this in a manner similar to the way we attempted to tackle message loops. When an outside event takes place and some procedure is triggered, we must trace the events that follow, to make sure that no procedure loop results. For each message that is created or modified or mailed, we must check if it triggers another procedure. The "chain of events" is therefore really a tree, starting with the first procedure triggered, and continuing along the paths of the messages handled by that and every subsequent procedure.

In the example of figure 5.6, the tree branches with the firing of $p$, since there are two messages output. $x$ must wait, so that branch terminates. $y$ triggers $p'$, which causes that branch to split into two. $y$ dies, and $z$ continues. Although branches of the tree may die, there is always some branch that continues, so the chain of events never terminates, and we have a procedure loop.

Let us now introduce the notion of a "*daemon*". A daemon is created when an outside event occurs, such as a user action. The event is given some unique identification (a time-stamp plus the nature of the event, for example). One copy of the daemon is assigned to each message affected by the event. We then see if these messages in turn trigger yet other procedures. If they do, then *each* of the outputs of the triggered procedure inherits the daemon. A daemon is thus "split" into copies whenever the chain of events branches.

In addition, we keep track of the history of each child daemon by remembering the sequence of procedures and message instances that daemon has passed through. We call this a *daemon path*. In our example we might start with:

$$(id,\ p)$$

This is inherited by $x$ and $y$, so we obtain:

$$(id,\ p\ x)$$
$$(id,\ p\ y)$$

The former dies and the latter continues. We eventually get a daemon of the form:

$$(id,\ p\ y\ p'\ z\ p\ x\ p'\ y\ p \cdots)$$

A daemon dies if the message it is currently associated with fails to trigger any new procedures. However, as long as one child daemon exists with a given identification, we know that the chain of events starting with the corresponding outside event has not yet terminated. Note that messages in message loops will be associated with the same daemon for the duration of the loop. An unmoderated triggering sequence therefore corresponds exactly to the tail of a daemon path in which the messages do not change. In the example of figure 5.5, the daemon path would be:

$$(id,\ p\ x\ p'\ x\ p\ x\ p' \cdots)$$

corresponding to the unmoderated triggering sequence $p\ p'\ p\ p' \cdots$.

How may we use daemons to detect procedure loops? Again we are faced with the fact that daemons cannot possibly tell us exactly when there is or is not a procedure loop, that problem being undecidable (as it was for message loops). We note, however, that a procedure loop exists if and only if some daemon can encounter at least one procedure infinitely often (clearly an unobservable event!). This suggests that we may detect *potential* procedure loops simply by noting when a daemon encounters some procedure more than once (or some bounded number of times). As with message loops, we could alert a user if we detect such an occurrence.

We are tempted to try to refine this technique. For example, we note that for a procedure loop to exist, the procedures must be able to indefinitely generate their own inputs. The obvious approach is to note for each message that waits, which daemon (i.e. which original event) was responsible for its current state. Whenever a daemon "dies" because the message it is associated with fails to trigger some procedure, we would continue to "remember" the daemon identification for future reference. Then, whenever a procedure is triggered, we can compare the daemon *id*s of the triggering message with those of the other messages. If all of the inputs are there as a result of some common original event, then we have some evidence that there may be a procedure loop.

For example, when $x$ waits at $i_2$ in figure 5.6 after the first firing of procedure $p$, we remember the daemon $id$. When message $z$ eventually catches up with it, we see that both $x$ and $z$ ultimately result from the same initial event.

There are two problems with this attempt at refinement. The first is that such evidence, albeit weighty, is *not* conclusive. If procedure $p$ generates two messages that are consumed by procedure $p'$, then the inputs to $p'$ will have the same daemon *id*s, though there is obviously no loop.

The second problem is that there is *not* necessarily a one-to-one correspondence between daemons and procedure loops. One daemon could easily result in several independent procedure loops. More

interestingly, two or more daemons may enter into a symbiotic relationship in which they feed message inputs to each other. In this case, some (possibly all) procedures in the loop would be triggered by inputs that do not have common daemon *id*s. To see this, consider the Petri net of figure 5.7.

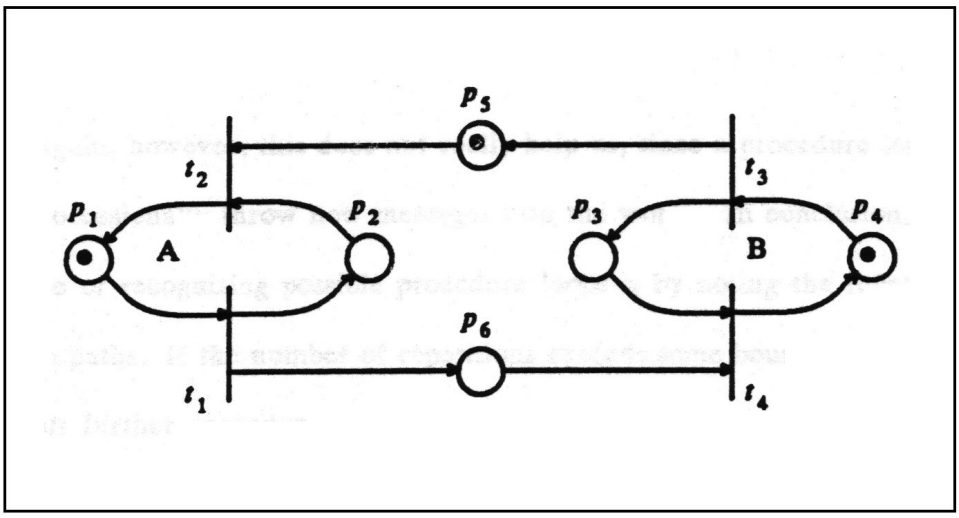



**Figure 5.7** : Symbiotic daemons

Here we have an initial marking of (1,0,0,1,1,0). Transitions $t_1$ and $t_3$ are enabled. Suppose that the tokens (representing messages) in places $p_1$ and $p_4$ have daemon *id*s A and B, respectively. The transition firing sequence $t_1\ t_2\ t_3\ t_4$ will bring us back to the initial marking, and so we have a procedure loop with that sequence repeated indefinitely.

Furthermore, the daemon for A remains alive, being associated with the token that travels between $p_1$ and $p_2$. During the entire sequence, either $t_1$ or $t_2$ is always enabled, so daemon A never dies. Only when A splits at $t_1$ does the child going to $p_6$ die, since it fails to trigger $t_4$. Similarly, daemon B never dies since one of $t_3$ or $t_4$ is always enabled.

In all fairness, we should mention that this unusual situation is very sensitive to timing. Suppose that A "got ahead of itself" and we had the firing sequence $t_1\ t_2\ t_1$. At this point we would have the marking (0,1,0,1,0,2). Neither $t_1$ nor $t_2$ would be enabled and daemon A would die entirely.

Daemon B could "take over" now by repeatedly firing the sequence $t_3\ t_4\ t_2\ t_1$. Daemon symbiosis is inherently unstable if the daemons truly rely on each other for input. One need simply "force" one daemon to exhaust itself by triggering procedures until it dies (after all, it depends on the another daemon), and then start up the other daemons again so they can "take over".

All of this may be entertaining, but it is not very encouraging for our search of procedure loops. This counter-example shows that it is unreasonable to expect that procedure loops will be limited to the case where inputs are ultimately generated by a single daemon only. We might attempt a further refinement by remembering all the daemons of matching inputs and noting whether we see the same faces repeatedly. For symbiosis to take place, some finite set of daemons must cooperate indefinitely. If we continue to see new daemons with newer and newer time-stamps, then we can be sure that we are not in a loop. If there *is* a procedure loop, however, then the finite set of daemons in the loop must have some newest daemon. That daemon should then never encounter coordinating inputs resulting from yet newer events.

Again, however, this does not really help us, since a procedure loop exists even if other events occasionally throw new messages into the works. In conclusion, then, the only means we have of recognizing possible procedure loops is by noting the repetition of procedures in daemon paths. If the number of repetitions exceeds some bound, then we may alert someone, and halt further procedure triggering until confirmation is received that it is alright to continue. If certain situations yield the spurious discovery of

procedure loops, one could increase the bound on the allowable number of procedure repetitions within daemons paths, or even disable the alerting facility entirely for procedures in that "loop".

## 5.5. Summary

In this chapter we have attempted to identify certain types of global behaviour that appear to be of special interest.

To aid our analysis we have shown how to recover the coordination that was temporarily lost when we obtained the message flow automata. These automata can be "joined together" to develop a Petri net interpretation that reflects both the message flow exhibited by the automata, and the procedure interactions that result from the coordination of messages in trigger conditions.

We discovered that non-determinism in our modeling effort has its sources not only in the systems that we are trying to analyze, but also to some extent in the way that we abstract messages into message states. The loss of information resulting from the grouping of messages into states can cause spurious non-determinism to appear in our model.

Blocking was also shown to have many potential causes. Messages cannot reach procedures awaiting them if there is no path to them. Procedures may also be blocked if there is a choice of paths, some of which avoid the procedure. Another possibility is that two procedures may each be waiting for a message that is stuck at the other. Finally, we have a recursive form of blocking in which messages cannot reach a procedure because they are blocked somewhere else.

Procedure loops are a potentially dangerous situation. We show that procedure loops may be detected by using our Petri net interpretation of message flow, and translating a matrix representation of our Petri net into a linear programming problem. The solution to the linear programming problem tells us whether or not there is a possibility of a procedure loop.

The information loss due to our use of message states makes the results of our analysis overly pessimistic in some cases. We would therefore like to be able to detect some problems at run time, independently of our message flow analysis. The most serious potential problem is that of procedure loops, since they can inflict much damage before they are detected by observation. We show how message loops and procedure loops can be caught early on by tracing the chain of uninterrupted events. Although we cannot conclusively prove that loops do or do not occur, we can at least identify what appears to be the start of one.

# 6. Concluding remarks

The aim of this research has been to develop techniques for formally describing and analyzing the global behaviour of a message management system. We have been interested in systems that provide some facility for its users to implement -- directly or indirectly -- procedures for automatically processing incoming messages. The interaction of these automatic procedures can be sufficiently complex to make it very difficult for anyone to tell how the system is behaving globally.

In our study of this problem, we have:

1. Developed a notation for a powerful model that captures the interactions of automatic procedures. The model allows us to represent workstations, mailboxes, a variety of message types and instances, and procedures that automatically process messages that match their trigger conditions.
2. We have developed the notion of message flow as a tool for characterizing global behaviour. Partitioning message domains into states allows us to express message flow with alternating sequences of message states and procedures. This yields a finite automaton interpretation of flow for individual message types, and a Petri net interpretation for procedure interaction.
3. We have classified types of message behaviour in terms of deadlock, "dead ends", message loops and procedure loops.
4. Algorithms for generating message state spaces, for collecting state transitions, and for detecting deadlock and procedure loops are presented.
5. In addition, techniques for the run-time detection of procedure loops are developed.

### 6.1. Limitations and possible extensions

Although our model allows us to capture fairly general procedures, there are some inherent limitations that make modeling of certain kinds of procedures difficult or awkward.

#### 6.1.1. Procedure inputs

Inputs to procedures, for example, are limited to a fixed number of messages of a fixed set of message types. Generally speaking, this assumption makes sense, since a procedure should be defined for a reasonably well-defined set of inputs. One might, however, want to be able to define a procedure that handles a variable number of order forms, for example. Presently one would have to do this by running a single procedure a variable number of times. A set of order forms could not be processed at the same time unless we defined several procedures (in our model) to handle each of the possible sizes of input sets. This makes it difficult to represent the idea of "dossiers", or messages that are electronically "stapled together".

Conditional creation of outputs is also not easily represented. Presently one must split such a procedure into two -- one to handle the situation in which the output is created, and the other to handle the case in which it is not. This is easily seen by referring to our Petri net representation. Petri net transitions do not conditionally output tokens. One must represent this behaviour with two (or more) transitions -- one for

each possible way of producing outputs. Extensions to the model that would allow variable numbers of inputs or conditional creation of outputs would therefore not help our Petri net interpretation since procedures would then still have to be multiply represented.

A similar extension would be to allow a choice of input types. This too must presently be represented with different procedures to handle the different combinations of input types. This would be useful if, for example, there are two different types of order forms and three kinds of inventory records. We would like to distinguish between them, up to a point. There would be six possible combinations of orders and inventory records, however, and each would have to be represented individually, if we wanted to maintain the distinction.

### 6.1.2. Specialization

The above problem may be seen as one of representing *specialization* of message types. (This is analogous to specialization in TAXIS. See chapter 2.) One message type would be the specialization of another if it has at least the attributes of other, and then some. A specialized order form would contain additional information, such as an area for special delivery instructions. (Another common notion of specialization is that of restricted attribute domains. Specialized messages would be limited to a restricted set of values.) A possible way of extending the model to allow for specialization of message types would be to represent the type and all its specializations as a single type with attributes that are the union of the attributes of all the subtypes. Non-specialized messages would have null values for the attributes that do not apply for that subtype. (No special delivery instructions for that order form.)

The advantage of this approach is that it solves the problem of representing procedures with a choice of message types for some inputs, provided the choices are all specializations of the same basic type. At that level we simply ignore the distinction. We can recapture that information at the level of partitioning message domains into state spaces: we would reserve special message states for attributes with null values. Groups of message states would therefore correspond exactly to given message subtypes. Procedures that are triggered by messages of some subtypes only could be represented using trigger conditions insisting that certain attributes have (or not have) null values.

### 6.1.3. Intelligent messages and objects

A far more interesting extension would be to allow for "intelligent messages" (as described in chapter 2). Currently we could only represent them by having procedures at all stations to capture the automatic behaviour of those messages. This has the disadvantage of disguising the fact that this behaviour is associated with the message type rather than with the workstations.

The distinction between procedures and messages gets very fuzzy if we notice that intelligent messages resemble procedures with memory that can be mailed from station to station. An extension to the model that would incorporate this idea, would probably replace procedures *and* messages by the more general notion of an *object* (as described in chapter 2). Objects would have one area for their memory that resembles the contents of a message, and another area that represents their behaviour. The behaviour could be made up of a set of rules resembling the triggers and actions of procedures. The major difficulty in modeling object interactions with Petri nets is that we no longer have a neat correspondence of message states and inputs to places, message instances to tokens and procedures to transitions. Furthermore, it is not clear that an object model helps our understanding of behaviour in message systems -- it is much more natural to draw a distinction between concrete objects (forms and records represented by messages) and abstract procedures for handling them (represented by activities). Perhaps some sort of hybrid would be appropriate, with some objects being identified as basically passive and message-like, and other being identified as more procedure-like.

## 6.2. Evaluating changes

One would like to be able to use the model to automatically evaluate changes to the system. Although we can characterize global behaviour before and after a change, we have not developed means for comparing the characterizations. There are two good reasons for this. The first is that our modeling

technique necessarily loses information. By abstracting messages into sets of values called "message states" we may mask precisely the effect of any changes. Secondly, even if we could rely on the message states as accurately capturing the global behaviour of the system we are modeling, we cannot effectively evaluate the change. Recall that our analysis results in a Petri net interpretation of procedure interactions. We would therefore have two Petri nets, representing the system before and after our change. Petri net equivalence is known to be an undecidable problem, however [Pete83].

On the other hand, Petri net equivalence is undecidable only for arbitrary nets. The changes that we are likely to want to evaluate will undoubtedly involve only a few procedures and station at most. Conceivably, one may be able to evaluate certain classes of small changes that are intended to not disrupt the overall behaviour of the system.

Recall that each message state represents messages at at most one location. Similarly procedures reside at given workstations, and may only accept inputs belonging to that station or one of its mailboxes. We may therefore partition our Petri net of message flow into a collection of subnets each representing the procedure interactions at one station. The subnets would be linked by procedures at one station that mail messages to other stations. Procedure output places at one station would therefore be linked to places of message states at other stations.

To evaluate the global consequences for a small change at one station, or at a small number of stations, one need only consider the corresponding subnets. If the subnet behaviour is "substantially the same" (i.e. no deadlocks or procedure loops are introduced), and if the connections between those subnets and others are not interrupted, then we may be assured that the global behaviour will not be radically altered.

It may be possible to do some of this analysis automatically by generating the new subnets, checking for deadlock and local procedure loops, and by checking the subnet interconnections. The final evaluation should nevertheless be carried out manually since the Petri net and its subnets carry incomplete information about global behaviour. The Petri net representation may, of course, be a useful aid in this analysis.

### 6.3. Message states

Another area for further investigation is the generalization of message states. Message states in this thesis may be thought of as "hypercubes", where the control attribute domains are the dimensions of the state-space. Since the attribute domains are partitioned into subdomains and ranges, the states are n-dimensional "boxes" or "hypercubes" (or sets of hypercubes).

This choice of message state partition is, of course, not the only possible one. In fact, such message states make it impossible to accurately represent trigger conditions involving more than one attribute. (If a trigger condition is expressible in disjunctive normal form as a composition of simpler conditions, it is these simpler conditions that we are interested in.) The hypercube message states are defined by conjunctions of simple conditions each involving at most one attribute. Message states defined by simple conditions that compare linear functions of attributes to some constant would define n-dimensional polyhedra (we are specifically considering numeric attributes here). Conditions like $x_i + x_j \leq 500$ would result in message states with very different "shapes".

In our analysis of message state transitions, we are faced with two problems. The first is determining what the image is of a given message state under the transformation defined by a procedure action and the possible coordinating messages. The second problem is to decide what other message states may possibly intersect that image. This will tell us what the state transitions are.

If we can solve these problems for differently shaped message states, then we can more accurately represent the messages that satisfy trigger conditions, and therefore more honestly capture global behaviour.

### 6.4. Related work

There is a strong analogy between operating systems and office systems. Messages can alternatively be viewed as resources for which procedures contend, or as jobs flowing from one processor (procedure) to another. *Flow expressions* [Shaw78] and *message transfer expressions* [Ridd73], can be compared to the

message flow expressions in this thesis.

Flow expressions are regular expressions for describing the flow of entities such as control, jobs and messages through programs, processes and other system software components. *Lock* symbols, and *wait* and *signal* symbols are embedded in these regular expressions to provide mutual exclusion and synchronization primitives. These expressions can then be *shuffled* together to indicate possible interleavings of the flows in each expression. The entities in the shuffled expressions must obey the locks, waits and signals in a concurrent execution. The shuffle operator generates the valid interleavings.

An alternative to generating Petri nets from message flow expressions to recapture coordination may be to introduce waits and signals. A message would generate a signal to indicate that it is in a certain state. Procedures, on the other hand, would have waits associated with their trigger conditions, and signals associated with their actions to represent the possible input and output message states. A particular system state could possibly be represented by a collection of signals, one for each message instance in the system. The shuffle of message flow expressions with embedded waits and signals would then yield the possible interleaved message flows with coordination.

Note that this approach would only work for single message instances of each type, since signals do not "stack". Two instances siumltaneously in the same state would effectively generate only a single signal. Since messages are not a limited, finite resource, it is not clear how useful waits and signals may be in representing message state transitions.

A number of interesting problems can be approached with flow expressions, including certain deadlock problems. Further research into flow expressions is reported in [Gisc81].

## 6.5. Other topics

There are several closely related topics that are not addressed by this thesis, but are of considerable interest.

One important issue is the design of systems that enable "naive" users to write their own automatic message-handling procedures. What should the power of these procedures be? Should their power be limited in any special ways? Should the specification of the procedures be limited in such a way as to more easily facilitate the analysis of global behaviour?

The systems described in chapter 2 suggest some trends, but these are still early approaches to the problem. Most of these systems are prototypes, and have not proven themselves in the marketplace. The assumptions inherent in our model may therefore not be truly representative of what is yet to come (as we mentioned earlier in this chapter).

Another important issue is performance analysis. Queueing network analysis is a possible approach. Messages can be thought of as jobs awaiting service at procedures. We do not, however, think of messages as being queued for service. Rather, they are serviced when combinations of messages appear that satisfy trigger conditions. Coordination does not traditionally have a place in queueing network models.

An approach that is perhaps more promising is that of stochastic Petri nets. They are used in [Mall81] to model communications networks. Some of Malloy's results may possibly carry over into the message system domain.

Related to performance analysis is the issue of restructuring the system. We may wish to maintain the current system behaviour, yet split a workstation into two to redistribute some of the load between several workers. The message flow model may be of use in deciding what transformations will guarantee the same overall behaviour. Rather than evaluating arbitrary changes, however, we would like to have some guaranteed ways of redistributing the load without having any net change. By analogy, we would like to have the same Petri net of system behaviour, but change the distribution of the subnets by moving some message states and procedures to different stations.

# Appendix A : Glossary

action :
: the functional part of a *procedure* that modifies or routes a message. The action takes place when the procedure's *trigger* condition is satisfied. $A(p)$ is the action of procedure $p$.

attribute :
: a field of a message. A *message domain* is assumed to be the Cartesian product of the attribute domains. The *location* and unique identifier of a message are two attributes of any message type. $X_{ij}$ is the $j$th attribute of $X_i$. $X_{i0}$ and $X_{i1}$ are reserved for the identifier and location of a message, respectively.

augmented Petri net :
: a *Petri net* in which each transition is associated with an additional set of rules, productions or preconditions and actions. See appendix C.

blocking :
: A message is blocked if it is waiting for processing at a procedure that is blocked. A procedure is blocked if one of its inputs will not arrive (i.e. the procedure cannot be triggered).

conflict :
: can occur in a *Petri net* when transitions share inputs. See appendix C. Both transitions may be enabled, but the firing of one may disable the other. Similarly, *procedures* that share input messages may conflict.

contents :
: the information contained in a message. The *location* is not usually considered part of the contents of a message.

control attribute :
: any message attribute that affects the routing of messages of that type. This includes *selection attributes*, routing attributes, and any attribute that is used in the action of a procedure to compute the new value of any control attribute.

daemon :
: a formalism for identifying the chain of events that results from some user action, or some other outside event. It is characterized by a unique identifier, and a *daemon path* consisting of the alternating sequence of procedures and messages encountered in the chain of events. Since one event may initiate several other events, we allow daemons to split, with each child inheriting the daemon path up to the splitting point. A daemon dies if no new event is triggered. A *procedure loop* exists if there are daemons that never die.

dead end :
: any *location* at which certain message instances get blocked because there is no procedure which it can potentially trigger.

dicomponent :
: a set of mutually reachable nodes in a directed graph. See also *condensation*.

enabled transition :
: a transition in a *Petri Net* that has at least one token in each input place. See appendix C.

input :
: any of the messages modified by a *procedure*. The list of types of the inputs of procedure $p$ is $I(p) = < I_{p1}, \cdots I_{pl_p} >$.

input tuple :
: a tuple of message values satisfying the *trigger* condition of a procedure. Often represented by $\tau$, where $x = \tau[j]$ is the $j$th message.

location :
: a unique attribute of any message instance. Every message has a location which is either a *mail box* or a *station*. The set of all locations is $L$. Every procedure is owned by a station and may be triggered by and modify only messages at that station or one of its mail boxes. $L(s)$ is the set of locations that procedures at $s$ may access.

mail box :
: the (temporary) *location* of a message that has been mailed from one *station* to another. Though there may or may not be true mail boxes implemented in a message management system, this formalism allows us to keep track of which messages have recently been mailed, and who their sender was. $m_{ij}$ is the mail box for messages sent from $s_i$ to $s_j$.

message :
: synonymous with *message instance*.

message class :
: an equivalence class of message instances that can potentially encounter the same sequences of locations and procedures. *Message states* are useful in identifying these classes.

message domain :
: the set of possible *message values* for messages of a given *message type*. The message domain is assumed to be the Cartesian product of the attribute domains. $dom(X_i)$ is the domain of messages of type $X_i$.

message flow expression :
: a regular expression that describes the non-deterministic finite automaton obtained by partitioning *message domains* into a state space and noting how procedures effect state transitions. A message flow expression approximates the *message paths* taken by messages in the same *message class*.

message flow language :
: the set of sequences of procedures a message $x$ may potentially encounter, denoted by $\hat{l}(x)$.

message instance :
: a message instance is an entity that may take on values from a *message domain*. The value is a tuple consisting of one value for each message attribute. To identify individual instances as they take on these values, we insist that each instance have one attribute which is a unique identifier.

message loop :
: a loop in the *message path*. If a message can attain the same value or value in the same equivalence class (*message class*) then that portion of its path may possibly be repeated indefinitely.

message management system :
: a computerized system for managing (in the sense of a database management system) structured messages and for automating some of the activities involving these messages.

message path :
: the alternating sequence of *message values* and *procedures* a *message instance* encounters from the time of its creation on.

message state :
: a block in a partitioned *message domain*. Two messages are deemed to be equivalent if they are in the same block or "message state". A finite automaton can then be constructed with *procedures* mapping states to states. Ideally the partition is made so as to identify messages in the same *message class*. Message states that work well in practice are obtained by partitioning the individual attribute domains and considering the Cartesian product of blocks in the attribute partitions.

message type :
: a descriptor for all messages chosen from the same *message domain*. The message type determines the attributes, attribute domains and, consequently, the message domain. $X = \{X_1, \cdots X_K\}$ is the set of all message types.

message value :
: an element in the *message domain*. A possible value held by a *message instance* consisting of the tuple of attribute values. The identifier, $x[0]$, of a message instance may never change.

office activity :
: a possibly long-term task to be accomplished in an office. An office activity is broken into individual steps called *office procedures*, or simply "procedures".

office information system :
: an integrated computer system that supports the functions and goals of an office.

office procedure :
: same as *procedure*.

ownership :
: indicates what station has control over mail-boxes, procedures and messages. Similar to *location*, except that messages whose location is a mail box are owned by the station owning that mail box.

path :
: See *message path*.

path equivalence :
: Two message are *path-equivalent* if they have the same *message flow language*. That is, they both may potentially encounter the same sequences of procedures, given the right coordinating messages.

Petri net :
: a formalism for modeling process synchronization. See appendix C.

place :
: a node representing a resource in a *Petri net* graph. See appendix C.

priority :
: a partial ordering on the set of procedures. If two procedures are enabled for input tuples including a common messages, then the "greater" procedure must fire first. This may disable the second procedure.

procedure :
: usually, synonymous with "office procedure". A procedure is a single step in a more complex *office activity*. A procedure is assumed to be atomic in the sense that it either runs to completion or does not run at all. Every procedure consists of *inputs* (the messages it is expected to transform), *trigger* conditions on its inputs, and *actions* tranforming those messages if or when the procedure is triggered. Since some procedures may not be completely algorithmic in nature, user input may be part of the procedure's actions. $P$ is the set of all procedures; $P(s)$ the set of procedures at station $s$.

procedure loop :
: an execution loop that can occur if a collection of procedures is capable of generating its own input. This is analogous to a *Petri net* that can fire forever See appendix C.

pseudo-station :
: stations that represent the creation or destruction of messages. These are denoted by $\alpha$ and $\omega$.

reachability set :
: the set of all markings of a *Petri net* that can be reached from a given marking by repeatedly firing *enabled transitions*. See appendix C.

routing attribute :
: any message attribute that is used in the action of a procedure to determine the next location of one of the input messages.

selection attribute :
: any message attribute that is used in the evaluation of the *trigger* condition of a procedure.

splitting :
: an instance of dividing a collection of messages into groups that will follow different *paths*. This may happen as a consequence of triggering different procedures or being routed in different directions.

splitting history :
: part of a *symbolic message*. The splitting history distinguishes the children of an original symbolic message by marking their position in a tree. The splitting histories can be examined to determine if all of the children have returned.

state :
: See *message state* and *system state*.

state transition :
: a triple ($p$, $\sigma$, $\sigma'$), such that procedure $p$ can map some message in *message state* $\sigma$ to state $\sigma'$. Also written $p: \sigma \mapsto \sigma'$.

station :
: a *location* with control over a collection of mail boxes, procedures and messages. Abbreviation for "workstation". $S = \{s_1, \cdots s_N\}$ is the set of all workstations.

symbolic message :
: A symbolic message is used to gather reachable state transitions. It simultaneously represents a set of message states, and it gathers transitions from those states as it travels from station to station. A symbolic message *splits* when transitions lead to different stations.

system state :
: the collection of all *message instances* and their current values at any given point in time. $D = < D_1, \cdots D_K >$ is the system state where $D_i$ is the set of instances of type $X_i$.

token :
: a dot in the place of a *Petri net* used to represent the availability of a resource. In a Petri net interpretation of message flow, a token is used to denote the presence of a message. See appendix C.

transition :
: a node representing a process in a *Petri net* graph. See appendix C.

trigger :
: a precondition on the firing of a *procedure*. A set of *input* messages that satisfies the trigger must be available. $T(p)$ is the set of *input tuples* satisfying the trigger.

unmoderated triggering sequence :
: a formalism for detecting *message loops*. Similar to a *daemon path*, except that it traces an unbroken chain of events for only a single message. The triggering sequence is the sequence of procedures encountered and triggered without any intermediate waiting. If such a sequence does not terminate, then we have a message loop.

workstation :
: synonymous with *station*.

# Appendix B : Notation

$A$ : $A(p): T(p) \to \prod_{j=1}^{l_p} dom(I_{pj})$ is the *action* of procedure $p$, mapping the messages of the input tuples in $T(p)$ to their new values. The input tuple $\tau$ is mapped to the output tuple $\tau' = A(p)(\tau)$. Individual attribute mappings are represented by $a_{jk}$, where $a_{jk}: \tau \mapsto \tau'[j][k]$. $A(p)$ may not change the *identity*, $\tau[j][0]$ of any message $\tau[j]$, i.e. $a_{j0}$ is always an identity mapping. Messages may only be routed to valid locations: $a_{j1}(\tau) \in R(s)$, where $p \in P(s)$.

$\hat{a}_p$ : $\hat{a}_p(x) = \{A(p)(\tau)[j] | \tau \in T(p),\ X_i = I_{pj},\ x = \tau[j]\}$ is the set of values that $x \in dom(X_i)$ may be mapped to after triggering $p$.

$D$ : $D = < D_1, \cdots D_K >$ is the *system state*. $D_i \subseteq dom(X_i)$ is the set of values of all currently existing message instances. There must be at most one message with any given identifier in $D_i$ (i.e. $\forall x \in D_i,\ y \in D_i,\ y[0] = x[0] \Rightarrow y = x$). $D(I)$ denotes $D_i$, where $I = X_i$ (this is useful when we do not wish to explicitly refer to the subscript $i$ of $X_i$).

$I$ : $I(p) = < I_{p1}, \cdots I_{pl_p} >$, where $I_{pi} \in X$, is the list of message types of the inputs to procedure $p$. $l_p$ is the number of inputs to $p$. One message of each type must be available to the procedure's local scope before the trigger condition can be evaluated.

$K$ : The number of message types, i.e. the size of $X$.

$L$ : $L = S \bigcup M$ is the set of all *locations* in the system, that is, all stations and mail trays. It is equal to $dom(X_{i1})$ for all message types $X_i$. $L^+ = S^+ \bigcup M$ includes the pseudo-stations $\alpha$ and $\omega$. $L(s_i) = \{\alpha, s_i\} \bigcup \{m_{ki} | 1 \le k \le N\}$ represents the *local scope* of station $s_i$ -- the locations from which procedures at $s_i$ may take messages. The $L(s) \backslash \alpha)$ partition $L$.

$\hat{l}$ :

$$\hat{l}(x) = \begin{cases} \{p\hat{l}(x') | p \in \hat{p}(x),\ x' \in \hat{a}_p(x)\} & \text{if } x_1 \neq \omega \text{ and } \hat{p}(x) \neq \varnothing \\ \lambda \text{ (the empty string)} & \text{otherwise} \end{cases},$$

the *message flow language* of $x$, is the set of possible sequences of procedures that $x$ may trigger. Two messages $x$ and $y$ are *path-equivalent* $\hat{l}(x) = \hat{l}(y)$. We write $x \tilde{} y$. We extend $\hat{l}$ to message states in the obvious way.

$M$ : $M = \{m_{ij} | 1 \le i \le N, 1 \le j \le N\}$ is the set of *mail trays* in the system, one for each pair of stations in $S$. Messages with location $m_{ij}$ have been sent from $s_i$ to $s_j$.

$N$ : The number of stations, i.e. the size of $S$.

$P$ : $P = \{p_{ij} | 1 \le i \le N, 1 \le j \le k_i\}$ is the set of all procedures in the system. $P(s_i) = \{p_{ij} | 1 \le j \le k_i\}$ is the set of procedures at station $s_i$, so $P = \bigcup_{i=1}^{N} P(s_i)$. $k_i$ is the number of procedures at $s_i$.

$\hat{p}$ : $\hat{p}(x) = \{p \in P | X_i \in I(p),\ X_i = I_{pj}, \exists\ \tau \in T(p)$ such that $x = \tau[j]\}$ is the set of procedures that may be triggered by $x \in dom(X_i)$.

$R$ : $R(s_i) = \{\omega, s_i\} \bigcup \{m_{ik} | 1 \le k \le N\}$ represents the valid locations to which procedures at $s_i$ may route messages.

$S$ : $S = \{s_1, \cdots s_N\}$ is the set of workstations, or simply *stations*, in the system. $S^+ = S \bigcup \{\alpha, \omega\}$ includes the *pseudo-stations* $\alpha$ and $\omega$ representing creation and destruction of messages.

$T$ : $T(p) \subseteq \prod_{j=1}^{l_p} dom(I_{pj})$ is the set of *input tuples* satisfying the trigger condition of $p$. In addition, if $\tau \in T(p)$, then $\forall j\ \tau[j][1] \in L(s_i)$ and $I_{pj} = I_{pk} \wedge j \ne k \Rightarrow \tau[j][0] \ne \tau[k][0]$ (the input messages must be in the local scope of $s_i$ and no message may play a duplicate role in the procedure). Tuple $\tau$ can thus *trigger* $p$ if $\tau \in T(p)$ and for all $I_{pj} \in I(p)\ \tau[j] \in D(I_{pj})$ or $\tau[j]$ is created by $p$.

$X$ : $X = \{X_1, \cdots X_K\}$ is the set of all *message types*. $X_{ij}$ is the $j$th attribute of type $X_i$. $dom(X_i) = \prod_{j=0}^{n_i} dom(X_{ij})$ is the domain of messages of type $X_i$. There are $n_i + 1$ attributes, including identity and location. $x \in dom(X_i)$ is the current *message value* of a message instance of type $X_i$. $x_j$ or $x[j]$ is the $j$th attribute of message $x$. $x_0$ is its *identity* and $x_1$ is its current *location*. Its identity never changes.

$\gamma$ : $\gamma(X_{ij}, p) = \bigcup \{arg(a_{kj}) | I_{pk} = X_i\}$ represents the set of attributes that affect the computation of any action in procedure $p$ that modifies attribute $X_{ij}$. It is used recursively to find the *control attributes* of message type $X_i$.

$\nu$ : $\nu = (s, \Theta, \Sigma, \chi)$ is a *symbolic message*. It is used to collect all message state transitions for a given message type starting at procedures at $s$. The state transitions are collected in $\Theta$. $\Sigma$ is the set of messages states currently represented by $\nu$. Symbolic messages may *split* if a state has two transitions leading in different directions. The *splitting history* $\chi$ identifies the children of the original symbolic message as nodes of a tree.

$\pi$ : a message state path, consisting of an alternating sequence of message states and procedures, for example, $\pi = \alpha\ p_1\ \sigma_1\ p_2 \cdots \sigma_n$.

$\rho$ :

$$\rho_{kp}(j) = \begin{cases} \{\tau \in T(p) | a_{k1}(\tau) = s_i\} & \text{if } j = 0 \\ \{\tau \in T(p) | a_{k1}(\tau) = m_{ij}\} & \text{if } 1 \le j \le N \\ \{\tau \in T(p) | a_{k1}(\tau) = \omega\} & \text{if } j = \omega \end{cases}$$

represents the set of input tuples to procedure $p$ for which the $k$th message is sent to station $s_j$. $\rho_{kp}(0)$ and $\rho_{kp}(\omega)$ are used to represent the case where message $k$ is not forwarded or is destroyed, respectively.

$\sigma$ : $\sigma \subseteq dom(X_i)$ is a *message state*, being a block of a partition of $dom(X_i)$. Also, if $\pi$ is a message state path, then $\sigma(\pi)$ denotes the last state in the string $\pi$.

$\tau$ : $\tau_p(\sigma) = \{\tau | \tau \in T(p),\ \tau[k] \in \sigma\}$ where $\sigma \subseteq dom(X_i)$ and $X_i = I_{pk}$) is the set of input tuples triggering $p$ that contain some message in state $\sigma$. For simplicity, $X_i$ and $k$ are understood.

$\chi$ : $\chi$ is the *splitting history* of a symbolic message. It distinguishes the children of an original symbolic message by marking their position in a tree. The history is a sequence of pairs $(j, n)$ each representing a node in the tree. $n$ is the number of branches at that node, and $j$ is the branch followed by that child. Splitting histories can be analyzed to tell if all the children of a symbolic message have terminated and returned.

$\phi$ :

$$\phi(\sigma) = \begin{cases} \{\sigma\ p\phi(\sigma') | p \in \hat{p}(\sigma),\ \hat{a}_p(\sigma) \bigcap \sigma' \ne \varnothing\} & \text{if } \sigma \ne \omega \text{ and } \hat{p}(\sigma) \ne \varnothing \\ \omega & \text{otherwise} \end{cases}$$

represents the message paths taken by messages in state $\sigma$. The strings in $\phi(\sigma)$ are alternating sequences of message states and procedures. They are reducible to $\hat{l}(\sigma)$ by mapping the procedures to the empty string.

$\gg$ : An optional *priority* may be placed on procedures to disambiguate conflict. If $p_i \gg p_j$ and both are enabled for overlapping input tuples, then $p_i$ must fire first. If $p_j$ is still enabled after $p_i$ no longer is, then it may fire.

# Appendix C : Petri nets

*Petri nets* are formalisms for modeling process synchronization and information flow [Pete83]. A Petri net is a directed bipartite graph. Nodes are either *transitions*, represented by bars, or *places*, represented by circles. Figure C.1 shows a simple Petri net.

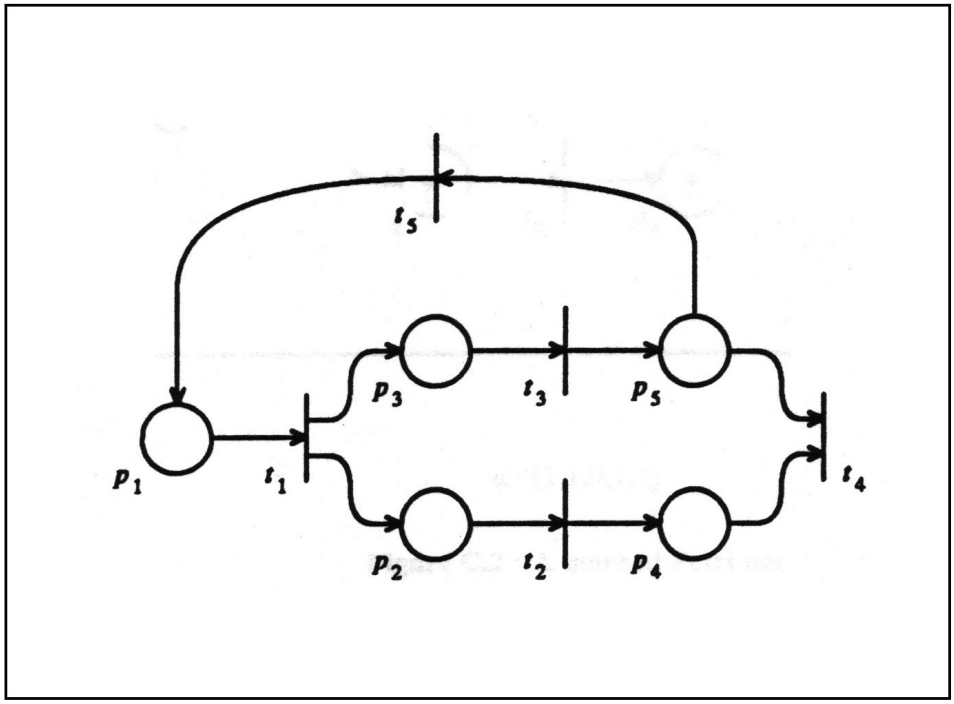
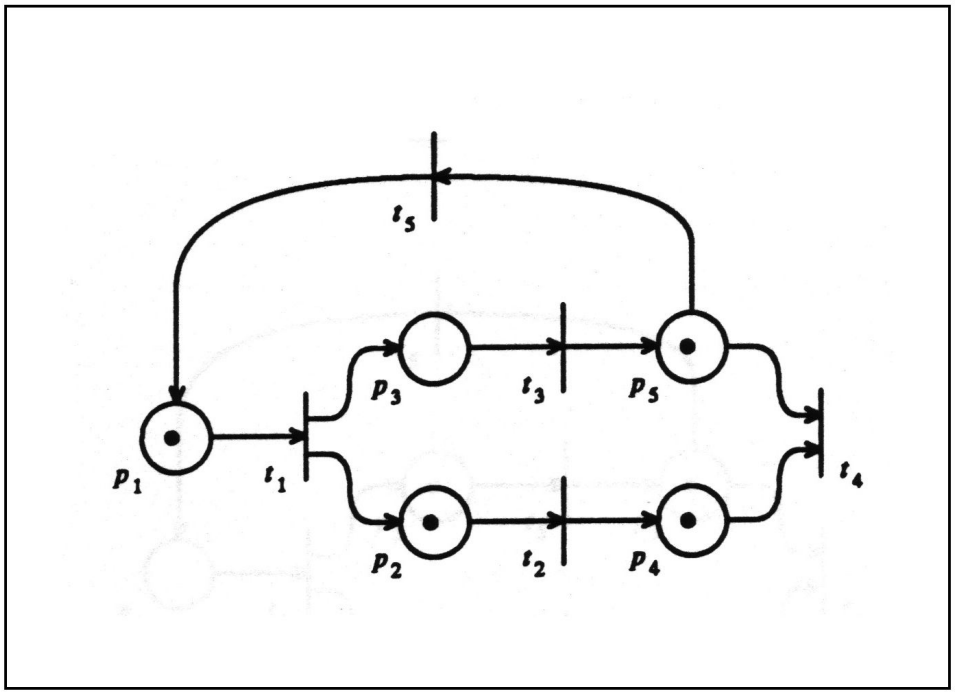



**Figure C.1** : A simple Petri net

Transitions usually represent processes and circles represent resources needed by the processes. We can formally represent a Petri net as $C = (P, T, I, O)$. $P$ is the set of *places* and $T$ is the set of *transitions*. $I$ and $O$ are both mappings from $T$ to $2^P$, the power set of $P$. $I(t)$ is the set of *inputs* of transition $t \in T$ and $O(t)$ is the set of *outputs* of $t$. There is an edge from place $p$ to transition $t$ if $p \in I(t)$ and there is an edge from $t$ to $p$ if $p \in O(t)$.

A *marked* Petri net is a Petri net with a number of *tokens* in each place, represented by black dots. A *marking* $\mu$, of a Petri net is a mapping from $P$ to the non-negative integers. ($\mu$ is usually represented by an $n$-vector, where $P = \{p_1, \ldots, p_n\}$ and $T = \{t_1, \ldots, t_m\}$.) Tokens indicate the availability of a resource or the completion of some event. The marking of a Petri net is its current "state". A marked Petri net is shown in

figure C.2.




$$\mu = (1, 1, 0, 1, 1)$$

**Figure C.2** : A marked Petri net

If there is a token in each input place of a transition $t$, then we say that $t$ is *enabled*. That is, $\mu(p) > 0$ for all $p \in I(t)$. Transitions $t_1$, $t_2$, $t_4$ and $t_5$, are enabled in figure C.2.

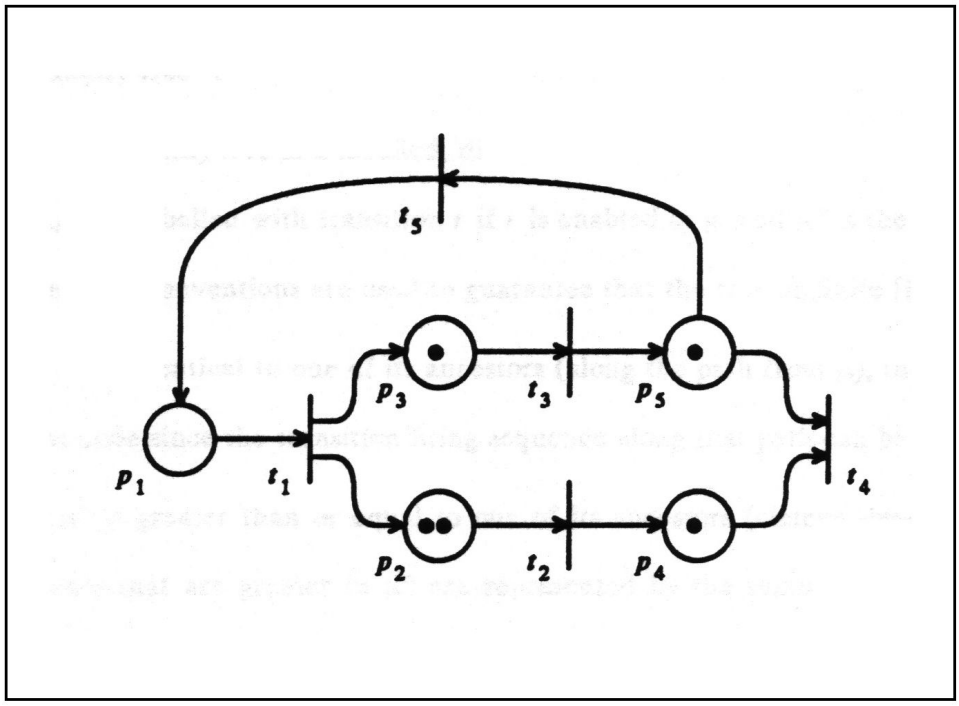
A Petri net may change state by *firing* enabled transitions. When a transition is fired, the Petri net acquires a new marking. If transition $t$ is enabled in marking $\mu$, then $t$ may be fired, and the Petri net changes state to $\delta(\mu, t) = \mu'$, where

$$\mu'(p) = \begin{cases} \mu(p) - 1 & \text{if } p \in I(t) - O(t) \\ \mu(p) + 1 & \text{if } p \in O(t) - I(t) \\ \mu(p) & \textit{otherwise} \end{cases}$$

This is represented graphically by removing a token from each input place of $t$ and adding a token to each output place of $t$. In figure C.3 we see the Petri net of figure C.2 after transition $t_1$ has been fired.

Petri nets are capable of modeling parallelism, contention for resources and coordination of events. In figure C.3, transitions $t_2$ and $t_3$ may fire in either order. After $t_1$ fires, $t_2$ and $t_3$ may execute in parallel. Transitions $t_4$ and $t_5$ compete for tokens in place $p_5$. Furthermore, since neither returns tokens to $p_5$, the firing of either $t_4$ or $t_5$ may disable the other transition (if both are enabled, and there are only enough tokens to fire one of them). Finally, we witness coordination in $t_4$, which can only fire if there are tokens in both $p_4$ and $p_5$. The parallel activities of $t_2$ and $t_3$ must both be completed.

Petri nets are typically non-deterministic in the sense that many firing sequences are possible from a given marking. The *reachability set* $R(\mu, C)$ of a Petri net $C$ is the set of all markings that can be reached from a given marking $\mu$ by repeatedly firing enabled transitions. Since there may be a choice of enabled




$$\mu' = (0, 2, 1, 1, 1)$$

**Figure C.3** : Transition $t_1$ is fired

transitions at any point, $R(\mu, C)$ must be generated in a non-deterministic fashion. $R(\mu, C)$ may be infinite. The Petri net in figure C.2 has an infinite reachability set since the firing sequence $t_1 t_3 t_5$ may be repeated indefinitely, causing the number of tokens in $p_2$ to become unbounded. We can characterize $R(\mu, C)$ by using a "reachability tree".

A *reachability tree* is a labelled, directed tree with some initial marking $\mu$ as its root. An arc $(\mu', \mu'')$ is labelled with transition $t$ if $t$ is enabled in $\mu'$ and $\mu''$ is the result of firing $t$. The following two conventions are used to guarantee that the tree be finite [KaMi69]:

1 if $\mu''$ is identical to one of its ancestors (along the path from $\mu$), then $\mu''$ is made a terminal node since the transition firing sequence along that path can be repeated indefinitely

2 if $\mu''$ is greater than or equal to one of its ancestors (element-by-element) then the elements that are greater in $\mu''$ are represented by the symbol $\omega$ (representing an integer that may be come arbitrarily large).

$R(\mu, C)$ is infinite if and only if the reachability tree contains the symbol $\omega$.

An equivalent representation of a Petri net is as a *vector addition system* [KaMi69]. $I$ and $O$ are represented by matrices $A^-$ and $A^+$ respectively, where the former consists of entries in $\{-1, 0\}$ and the latter of entries in $\{0, 1\}$. Both are $n$ by $m$ matrices, with one column for each transition. The $j$th entry of column $A^+{}_i = 1$, for example, if and only if $p_j \in O(t_i)$. The matrix $A = A^- + A^+$. For the Petri net of figure C.1,

$$A = \begin{bmatrix} -1 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 & -1 \end{bmatrix}$$

Many ideas can now be easily expressed in matrix terms. Transition $t_i$ is enabled in marking $\mu$ if $\mu + A^-_i \geq 0$ (element-by-element). The result of firing $t_i$ is $\mu' = \mu + A_i$. The result of firing a sequence of transitions is $\mu' = \mu + Ax$ where $x$ is an $m$-vector and $x_i$ is the (integral) number of times $t_i$ is fired in the sequence.

Variations on Petri nets permit multiple arcs between pairs of places and transitions, *inhibitor arcs* that inhibit the firing of a transition if the connected place contains any tokens, and others.

Petri nets with exactly one input and one output per transition are finite state machines. They can be viewed as finite automata by considering the transitions as *next-state* functions mapping places to places. A single token is needed to represent the initial state of the automaton. The strings accepted are the transition firing sequences (or equivalently alternating sequences of transitions and places, since transitions have unique outputs).

An *augmented Petri net* is a Petri net in which each transition is associated with an additional set of rules, productions or preconditions and actions.

# D. Bibliography and references


[AhHU74] A.V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[AtBS79] G. Attardi, G. Barber and M. Simi, "Towards an Integrated Office Work Station", AI Laboratory, MIT, Cambridge, 1979.

[Barr82] John L. Barron, "Dialogue and Process Design for Interactive Information Systems using Taxis", *Proceedings ACM SIGOA*, pp. 12-20, Philadelphia, June 1982.

[BoMu76] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, North Holland, New York, 1976.

[BySJ82] Roy J. Byrd, Stephen E. Smith and Peter de Jong, "An Actor-Based Programming System", *Proceedings ACM SIGOA*, pp. 67-78, Philadelphia, June 1982.

[BYTE81] Special issue on Smalltalk, Byte, 6(8), Aug 1981.

[Cheu79] C. Cheung, *OFS -- A Distributed Office Form System with a Micro Relational System*, M.Sc. thesis, Department of Computer Science, University of Toronto, 1979.

[Cook80] C.L. Cook, "Streamlining Office Procedures -- an Analysis using the Information Control Net Model", *Proceedings of the NCC 1980*, pp. 555-565.

[deBy80] Peter de Jong and Roy J. Byrd, "Intelligent Forms Creation in the System for Business Automation", IBM Research Report, RC 8529, Yorktown Heights, N.Y., 1980.

[deJo80] Peter de Jong, "The System for Business Automation: A Unified Application Development System", *Proceedings of IFIP Congress 80*, pp. 469-474, Tokyo, 1980.

[deZl77] Peter de Jong and Moshe Zloof, "The System for Business Automation (SBA): Programming Language", *Communications of the ACM*, 20(6), pp. 385-396, June 1977.

[Eile74] S. Eilenberg, *Automata, Languages and Machines*, Volume A, Academic Press, New York, 1974.

[Elli79] Clarence A. Ellis, "Information Control Nets", *Proceedings of the ACM*, Conference on Simulation, Measurement and Modification, Boulder, Colorado, pp. 225-240, Aug 1979.

[ElNu80] Clarence A. Ellis and Gary J. Nutt, "Computer Science and Office Information Systems", *ACM Computing Surveys*, pp. 27-60, March 1980.

[FiHe80] R.E. Fikes and D.A. Henderson, "On Supporting the Use of Procedures in Office Work", MIT workshop, Cambridge, 1980.

[Geha82] N.H. Gehani, "The Potential of Forms in Office Automation", *IEEE Transactions on Communications*, 30(1), pp. 120-125, Jan 1982.

[Gibb79] Simon Gibbs, *OFS: An Office Form System for a Network Architecture*, M.Sc. thesis, Department of Computer Science, University of Toronto, 1979.

[Ginz68] A. Ginzburg, *Algebraic Theory of Automata*, Academic Press, New York, 1968.

[Gisc81] Jay Gischer, "Shuffle Languages, Petri Nets, and Context-Sensitive Grammars", *Communications of the ACM*, 24(9), September 1981.

[HaKu80] M. Hammer and J.S. Kunin, "Design Principles of an Office Specification Language", *Proceedings of the NCC*, pp. 541-547, 1980.

[HaSi80] M. Hammer and M. Sirbu, "What is Office Automation?", *Office Automation Conference*, Georgia, pp. 37-49, 1980.

[HeBa77] Carl Hewitt and H. Baker, "Laws for Communicating Parallel Processes", *Information Processing 77*, ed. G. Gilchrist, pp. 987-992, North-Holland, 1977.

[Hewi77] Carl Hewitt, "Viewing Control Structures as Patterns of Passing Messages", *Artificial Intelligence*, 8(3), pp. 323-364, June 1977.

[HHKW77] M. Hammer, W.G. Howe, V.J. Kruskal and I. Wladawsky, "A Very High Level Programming Language for Data Processing Applications", *Communications of the ACM*, 20(11), pp. 832-840, Nov 1977.

[HMGT83] John Hogg, Murray Mazer, Stelios Gamvroulas and Dennis Tsichritzis, "Imail, an Intelligent Mail System", *IEEE Database Engineering*, 6(3), pp. 36-42, Sept 1983.

[Hogg81] John Hogg, *TLA: A System for Automating Form Procedures*, M.Sc. thesis, Department of Computer Science, University of Toronto, 1981

[Holt72] Richard C. Holt, "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys*, 4(3), pp. 179-196, September 1972.

[HoUl79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[KaMi69] R.M. Karp and R. Miller, "Parallel Program Schemata", *J. Computer and Systems Science 3*, pp. 167-195, May 1969.

[LaTs80] Ivor Ladd and Dennis Tsichritzis, "An Office Form Model", *Proceedings AFIPS NCC*, Anaheim, California, pp. 533-540, 1980.

[Loch83] F. Lochovsky, "Improving Office Productivity: A Technology Perspective", *Proceedings of the IEEE*, 71(4), pp. 512-518, April 1983.

[LuYa81] D. Luo and S.B. Yao, "Form Operation By Example", *Proceedings of the ACM SIGMOD Conference*, Ann Arbor, pp. 212-223, 1981.

[Mall81] Michael Karl Malloy, *On the Integration of Delay and Throughput Measures in Distributed Processing Models*, PhD dissertation, University of California, Los Angeles, 1981.

[MiYa77] R. Miller and C.K. Yap, "Formal Specification and Analysis of Loosely Connected Processes", IBM Research Report #28917, Yorktown Heights, N.Y., 1977.

[Morg78] Howard L. Morgan, "Control and Tracking of Office Documents", MIDCON Proceedings, Dallas, Texas, 1978.

[Morg80] Howard L. Morgan, "Research and Practice in Office Automation", *Proceedings 1980 IFIP Congress*, pp. 783-789.

[MyBW80] John Mylopoulos, P.A. Bernstein and H.K.T. Wong, "A Language Facility for Designing Interactive Database-Intensive Applications", *ACM TODS*, 5(2), pp. 185-207, June 1980.

[Nier81] Oscar M. Nierstrasz, *Automatic Coordination and Processing of Electronic Forms in TLA*, M.Sc. thesis, Department of Computer Science, University of Toronto, 1981.

[NiMT83] Oscar M. Nierstrasz, John Mooney and Ken Twaites, "Using Objects to Implement Office Procedures", *CIPS conference proceedings*, Ottawa, pp. 65-73, May 1983.

[PaSt82] Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization*, Prentice-Hall, 1982.

[Pete77] James L. Peterson, "Petri Nets", *ACM Computing Surveys*, 9(3), pp. 223-252, Sept 1977.

[Pete83] James L. Peterson, *Petri Nets Theory and the Modeling of Systems*, Prentice-Hall, 1983.

[Pott78] D. Potts, *Specifications Language for Office Procedures Execution*, Thesis, The Wharton School, University of Pennsylvania, Philadelphia, 1978.

[Ridd73] William E. Riddle, "A Method for the Description and Analysis of Complex Software Systems", *ACM SIGPLAN Notices*, 3, pp. 133-136, Sept 1973.

[Ruli79] J.F. Rulifson, "Information Systems Management and Misapplied Methodologies", Interoffice memo, XEROX PARC, March 21, 1979.

[Shaw78] Alan C. Shaw, "Software Descriptions with Flow Expressions", *IEEE Transactions on Software Engineering*, SE-4(3), pp. 242-254, May 1978.

[SSKH82] M. Sirbu, S. Schoichet, J. Kunin and M. Hammer, "OAM: An Office Analysis Methodology", in *Office Automation Conference 1982 Digest*, pp. 317-330, AFIPS, 1982.

[TRGN82] Dennis Tsichritzis, Fausto Rabitti, Simon Gibbs, Oscar M. Nierstrasz and John Hogg, "A System for Managing Structured Messages", *IEEE Transactions on Communications*, 30(1), pp. 66-73, January 1982.

[Tsic82] Dennis Tsichritzis, "Form Management", *Communications of the ACM*, pp. 453-478, July 1982.

[Zism77] M.D. Zisman, *Representation, Specification and Automation of Office Procedures*, PhD thesis, Wharton School, University of Pennsylvania, Philadelphia, 1977.

[Zloo80] Moshe M. Zloof, "A Language for Office and Business Automation", in *Proceedings of the AFIPS Office Automation Conference*, March 1980.