12. Message Flow Analysis

O.M. Nierstrasz

ABSTRACT

Message management systems with facilities for the automatic processing of messages can exhibit anomalous behaviour such as infinite loops and deadlock. In this paper we present some methods for analyzing the behaviour of these systems by generating expressions of message flow from the procedure specifications. Message domains are partitioned into state spaces, and procedures can be interpreted as automata effecting state changes. Blocking of procedures and procedure loops can then be detected by studying the resulting finite automaton and Petri net representations of message flow.

1. Overview

Automatic processing and routing of electronic documents yields some interesting problems when the work that is done with them is sufficiently complicated. In this paper we consider the task of determining what global behaviour is exhibited by messages in a message management system when there exist a number of automatic procedures running at user workstations, examining, processing and routing incoming messages.

If the logic built into these procedures is anything but entirely routine, then we may see messages being routed through the system in various ways. If the automatic procedures are adapted from existing manual procedures, there is always a possibility that the translation will be faulty: that messages may get improperly routed, or that procedures will wait indefinitely for messages that do not arrive. We therefore propose some techniques for studying and analyzing the behaviour that can be expected to result from such automatic procedures. The intended behaviour can thus be verified to some degree, and anomalous behaviour can be detected in advance.

In the following section we describe informally the systems that we are interested in modelling and analyzing. Collections of workstations connected by a network are used to pass electronic documents, or "messages". These messages are typically highly-structured, and often resemble forms. Similar messages are classified into "message types". High-level automatic procedures may in fact be implemented by the workers using the workstations. Complex activities can be broken down into simple steps that collect a set of messages satisfying "trigger conditions", perform transformations on those messages, possibly creating or destroying some, and then route or file them.

In the third section we introduce a formal model for discussing these systems. The model is then used to develop a characterization of global behaviour in terms of message flow. The message domains (the sets of values that messages may assume) are partitioned into state spaces. Procedures can then be viewed as effecting state transition on messages, and the entire system can be viewed as a collection of finite state automata, one per message type. We then show how to recover the coordination of messages performed by the automatic procedures by "welding" the finite state automata into a Petri net (a popular modelling tool).

Sections six and seven are concerned with detecting anomalous behaviour. In section six we discuss the problem of blocking, in which a procedure may wait indefinitely for a missing message to arrive. This is especially troublesome if there are other messages waiting to be processed by that procedure. There are various scenarios in which blocking may occur, including *deadlock*, where two procedures are each waiting for messages that are stuck at the other procedure.

In *Office Automation: Concepts and Tools*, D. Tsichritzis (Ed.), pp. 283-314, Springer-Verlag, Heidelberg, 1985.

In section seven we discuss "procedure loops". Here we may see procedures firing indefinitely, passing messages back and forth between them. A special case is the "message loop", in which some messages visit the same sequence of procedures indefinitely. These problems may also cause blocking, if a procedure is waiting for a message in a loop. If messages are created in the loop, the file system will eventually get saturated, and the network may even get overloaded with message traffic. We show how it is possible to use the Petri net model of message flow to detect possible procedure loops.

2. Message Management

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 by 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 [AtBS79, ElNu80, HaSi80]. 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.

The static objects in these systems 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").

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.

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. All messages satisfying a simple constraint can be automatically forwarded to a particular location. Order forms for large amounts could be forwarded to a manager for approval.

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.

3. Message Flow Modelling

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.

3.1. 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*, α and ω, that represent creation and destruction of objects. Creation and destruction are thus explicitly modelled. 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 α and that none be received from ω. That is, they must behave as *source* and *sink*, respectively. The set of stations and pseudo-stations is:

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

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 modelling, 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 modelling 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 α and ω do not have mailboxes. A message "from" α appears at the station creating the message. A message that is destroyed goes directly to ω. A station is allowed to mail messages to itself.

The set of all locations is

$$L = S \cup M$$

and, with the pseudo-stations:

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

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

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

This is the *local scope* of s_i — the locations that are accessible to the procedures at s_i. Messages may be created at α, 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:

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

(Note the reversal of subscripts on the mailboxes.)

3.2. 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 α or ω is not explicitly represented). A message value is represented by

$$x \in dom(X_i)$$

The kth attribute of x is denoted by either x_k or $x[k]$. The latter notation is generally used when x is the jth 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 α or ω. 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.

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

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).

3.3. 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:

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

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 τ, where $x = \tau[j]$ is the jth input message and $x_k = \tau[j][k]$ is the kth attribute value of the jth message. Such a tuple τ 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 τ must be available to p, that is, $\tau[j][1] \in L(s_i)$, and each of the messages in τ must be unique (a message cannot 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 τ 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 jth 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 α and ω. They allow us to (somewhat artificially) model messages that have not been created as arriving from α, and those that are destroyed as being sent to ω.

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}s 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 *set* of possible values (modulo the outside information sources). 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 τ 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:

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

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

4. Message Paths and States

Our model of message management views procedures and locations as 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, through 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".

Unfortunately this expression of message flow is impractical. In [Nier84] it is shown that there is no effective way of comparing the message paths of two different messages. Briefly, it is shown how two messages can "simulate" two different Petri nets in such a way that the message paths are equivalent to the Petri net languages. Since there is no effective way of determining whether two Petri net languages are equivalent [Pete83], we cannot compare message paths.

We must therefore seek some less demanding way of describing message flow. By partitioning message domains into finite state spaces 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.

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. (There are situations where this may not be so, but we shall not discuss them here.) 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 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.

Control attributes can be determined by a recursive application of the definition given above. Once the routing and selection attributes are determined, it is a relatively straightforward operation to detect the control attributes. An algorithm for doing this is described in [Nier84].

4.1. 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.

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.

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 $\vee(\wedge C_j)$ 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 $\vee(\wedge C_j)$, one should use the conjunctions $\wedge C_j$ to deduce the simple conditions.

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 τ 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.

4.2. 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 σ we would like to know the possible next state, σ', 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 σ:

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

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

We also introduce $\hat{p}(\sigma)$ as the set of procedures that σ might trigger, and $\hat{a}_p(\sigma)$ as the set of values that σ might be mapped to after triggering p:

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

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

Procedure p then effects a state transition from σ to σ' if $p \in \hat{p}(\sigma)$ and $\hat{a}_p(\sigma) \cap \sigma' \neq \emptyset$. That is $p: \sigma \rightarrow \sigma'$ if p is capable of mapping some message in state σ to some message in state σ', given the right coordinating messages. We also introduce $\hat{l}(\sigma)$ as the set alternating strings of message states and procedures encountered by messages starting in state σ:

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

$\hat{l}(\sigma)$ therefore is the message flow language for message state σ. It represents all sequences of procedures that messages in state σ 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:

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

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 Σ_i be the set of message states for message type X_i, i.e. Σ_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 α, the final state ω, and the next-state function is:

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

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

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. Such an algorithm is described in [Nier84].

Briefly, "symbolic messages" gather all the reachable state transitions by simply traversing a "spanning tree", starting at α, and visiting each station where the information about the transitions resides. A symbolic message 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.

When there are no new states and state transitions to visit, the symbolic message 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].

5. Petri Net Representation

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 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 σ_1 through σ_4 and σ'_1 through σ'_5 are represented by places. Petri net transitions are also present to represent the fact that input i_1 corresponds to message states σ_1 and σ_2, and that p generates outputs in state σ_4. An entire Petri net may be built in this way with transitions mapping message states of various types to other message states.

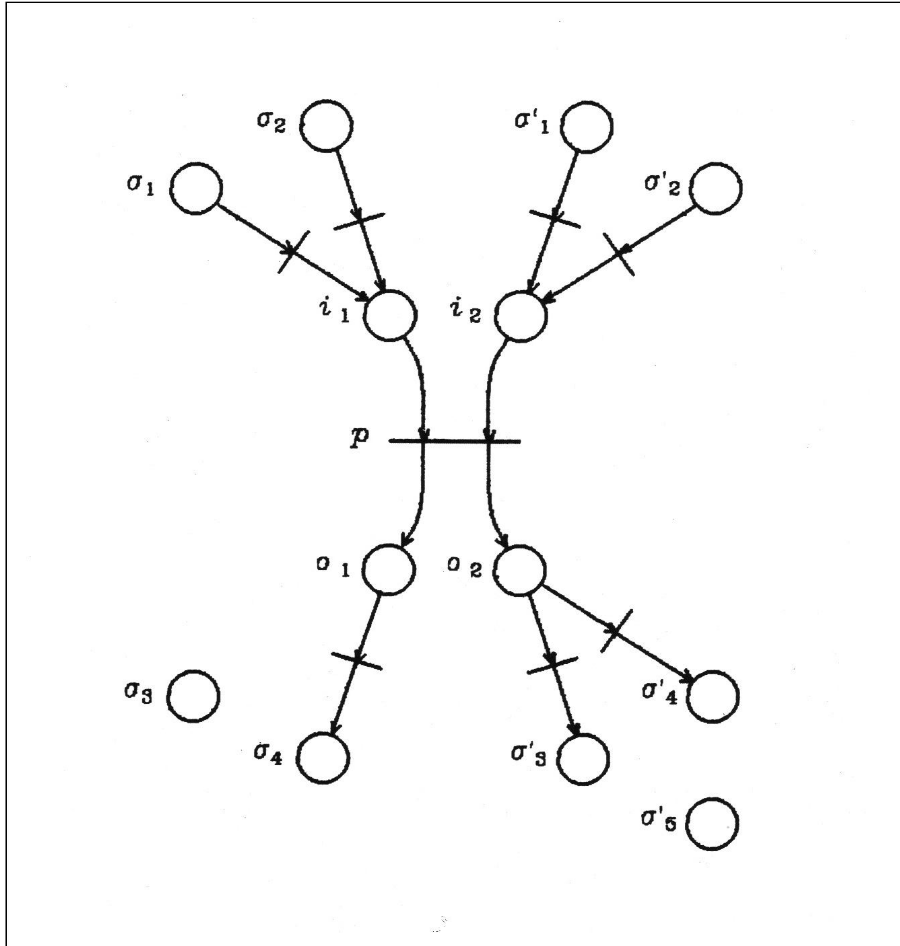

Figure 1: A Petri net interpretation of message flow

There is a serious problem here, however. In figure 1 it appears that messages in states σ'_1 or σ'_2 may map to messages in states σ'_3 or σ'_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 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. Σ_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, far fewer). Also, let $r_i = \{(p, \sigma_j) | \exists\ \sigma_k$ such that $(p, \sigma_j, \sigma_k) \in T_i\}$. The r_is represent the σ_js that trigger some procedure p. We shall use the elements of these sets as labels for the places and transitions of our Petri net.

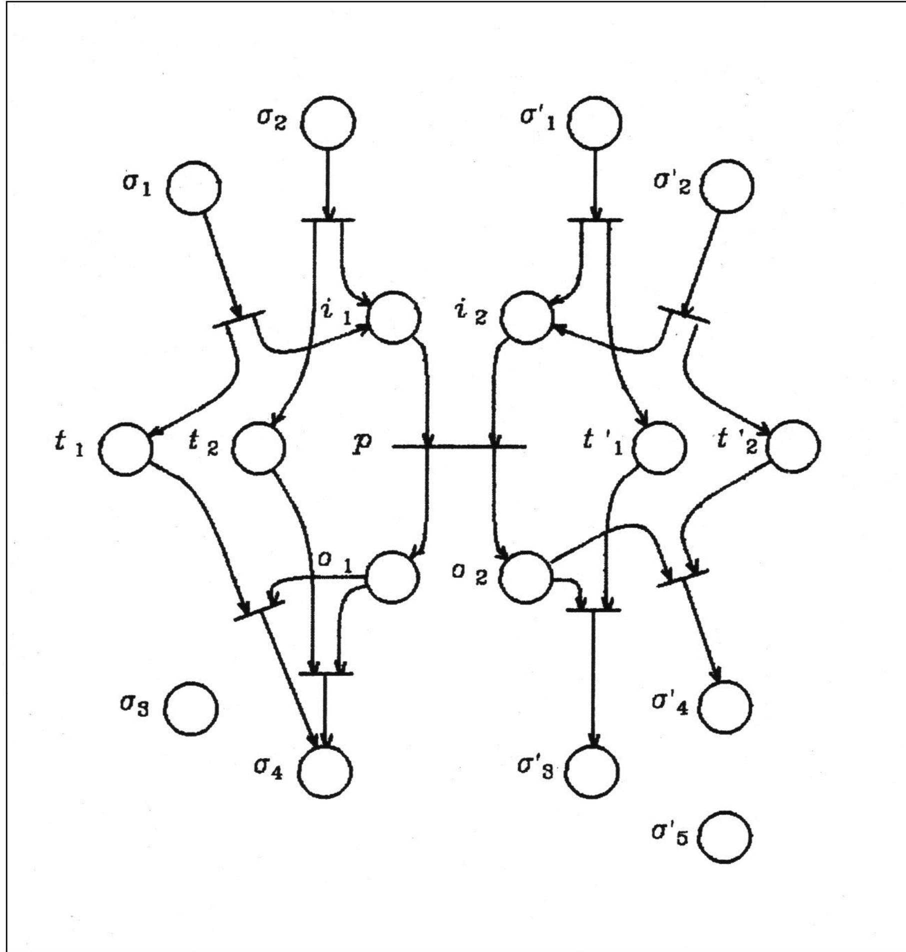

Figure 2: An "improved" Petri net interpretation

Let our Petri net have places with labels in:

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

and transitions with labels in:

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

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 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 σ_j, and has outputs (p, σ_j) and I_{pk} where $I_{pk} = X_i$
3. a transition labeled $(p, \sigma_j, \sigma_k) \in T_i$ has inputs (p, σ_j) and O_{pk} where $O_{pk} = X_i$, and has output σ_k.

It is now clear from the construction that tokens may "travel" from message state σ_j to state σ_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 the previous section.

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 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 α state or an ω state. Tokens could then be added at will to the α states, and removed from the ω states. Equivalently, we may simply delete procedure input and output places corresponding to the creation or destruction of messages. Message states α and ω need not be explicitly represented in this case.

6. Blocking and Deadlock

A procedure is *blocked* if it waits indefinitely for one of its inputs to arrive. If the procedure has only one input, 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.

6.1. 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.

6.2. 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 as we collect the state transition information, since only reachable states are encountered. 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 as we collect the state transitions.

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, ω 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 them for another dicomponent.

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

6.3. 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 2, message states σ_1 *and* σ_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 α, and avoiding all nodes that are input message states to that procedure. If we can construct a path to ω 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*.

6.4. 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:

$$X_i \xrightarrow{*} X_k$$

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 $X_i \xrightarrow{*} X_k$, then messages of type X_i must await messages of type X_k, since the latter are uniquely determined by the former.

If $X_i \xrightarrow{*} X_j$, *and* $X_j \xrightarrow{*} X_i$, (i.e. $X_i \xrightarrow{*} X_i$) 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*. $X_i \xrightarrow{*} X_i$ 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 $X_i \xrightarrow{*} X_j$, 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*.

6.5. 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:

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

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: $X_i \to X_j$ **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 σ is a new BL-state **do** {
9.   **for each** input $X_j \in I(p)$ **do** {
10.   **if** p: $X_i \to X_j$ **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.

7. 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 unending 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 earlier discussion of message loops 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 3. 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.

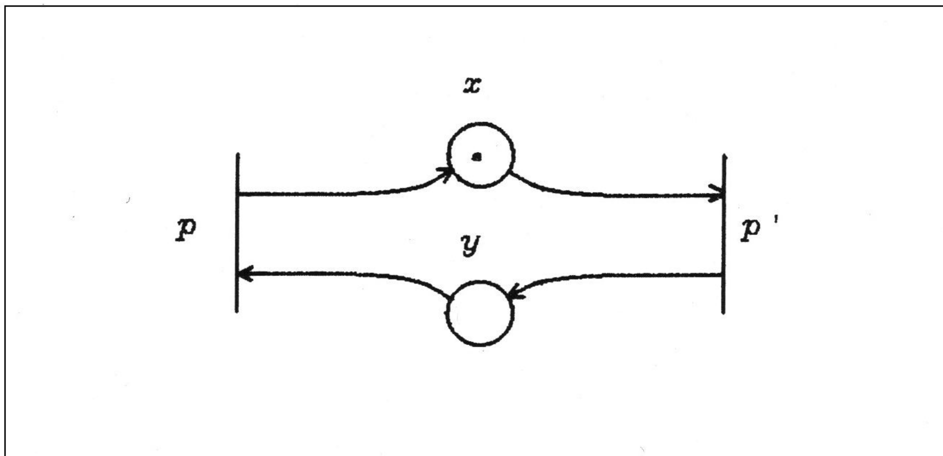

Figure 3: A procedure loop

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 μ is a marking of the Petri net, and $t_1 \cdots t_n$ is a transition firing sequence yielding new marking μ', then $t_1 \cdots t_n$ can be repeated infinitely often if $\mu_i \leq \mu'_i$ for each i.

We approach the problem of detecting procedure loops by translating it into an equivalent problem expressible in matrix equations. 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 3, 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 μ if $\mu + A^-{}_j \geq 0$ (where $A^-{}_j$ is the jth 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 μ, 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 μ, 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 μ. Furthermore, we can always find a marking μ "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 that remains is whether or not we can easily solve $Ax \geq 0$. To this end we present the following theorem:

Theorem : 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 fires, 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:

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

Now

$$A' x'' + A' x''' \geq (1, 0, \ldots, 0)^T$$
$$A' x'' \geq (1, 0, \ldots, 0)^T - A' x'''$$
$$A' x'' \geq (1 - x'_0, 0, \ldots, 0)^T$$

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 an 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. □

8. Conclusions

We have presented a formalism for modelling message systems with automatic processing of messages, and we have introduced some concepts that are useful in characterizing the global behaviour of these systems. We have shown how to generate finite state automaton and Petri net interpretations of message flow by using our model. Finally, we have shown how these derived interpretations can be useful in analyzing message behaviour. In particular, procedure loops and various kinds of blocking (including deadlock) can be detected.

A number of extensions to the model would be desirable. Messages are currently very simple. There is no explicit way of representing repeating groups within messages, nor do we explicitly handle "specializations" of message types. Similar and related (but non-identical) message types must therefore be treated as being distinct. We also do not currently allow procedures to handle inputs with a choice of input types. (One way to handle specializations, however, is to model them with a single "master" type combining the attributes of all the specializations, and simply assign null values to the inapplicable fields of particular message instances.)

A more radical extension is to allow for "intelligent messages" that carry procedures around with them. Procedures are currently associated with workstations, and not messages. An alternative is to consider the behaviour of a system that manages "objects", where an object combines the data-storing of messages and the functionality of procedures. It is not at all clear, however, how one would begin to analyze object-flow, once the distinction between data and procedure is lost.

Other interesting issues are the evaluation of incremental changes to systems, and the evaluation of transformations. In the first case we only make small, occasional changes such as adding or altering

procedures, and in the latter case we may coalesce or split workstations, or move procedures from one workstation to another. What questions are appropriate to ask about the effect of such changes, and can we make cheap evaluations based on the analysis of the unchanged system?

9. 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.

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

[ElNu80] C. Ellis and G. Nutt, "Computer Science and Office Information Systems", *ACM Computing Surveys*, 12(1), pp. 27-60, March 1980.

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

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

[Nier84] O.M. Nierstrasz, *Message Flow Analysis*, Ph.D. thesis, Department of Computer Science, University of Toronto, CSRI Technical Report #165, 1984.

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

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