4

Message Flow Control

A.T.M. Aerts[1], *W. Hofman*[2], *L.J. Somers*[1]
[1]*Dept. Math. and Comp. Sci., Eindhoven Univ. of Technology,*
P.O. Box 513, 5600 MB Eindhoven, The Netherlands,
+31 40 472733, 436685 (fax), {wsinatma,wsinlou}@win.tue.nl
[2]*Bakkenist Management Consultants,*
P.O. Box 23103, 1100 DP Amsterdam, The Netherlands

Abstract

High level communication by means of the exchange of messages between different organizations or organizational units is studied. The concepts of business transaction and business transaction protocol are discussed. Message exchanges are controlled by a message handler that enforces a transaction protocol. The design and implementation of a generic message handler is described. The design is independent of a specific application. It offers a flexible implementation of protocols: dedicated subprocesses handle each message type. Changes in a protocol can easily be accommodated by adapting the subprocesses. A prototype has been written in the executable specification language ExSpect. The protocols can be validated by simulating the message exchange for a number of cases. As an application of the prototype, a party in a logistic chain using EDI is described.

Keywords

Business transactions, Transaction protocol, Simulation, EDI, Petri nets

1 INTRODUCTION

When organizations grow and become decentralized, the need for communication between the units increases. A similar situation arises when different organizations have a stable and intensive business relation. Communication between these organizations, which we will regard as parties or actors in a communication network, will have to satisfy high standards with regard to such aspects as timeliness and correctness.

Electronic interchange of normalized and standardized data (messages) between the (automated) information systems of different parties, EDI for short, has become an important technique for the implementation of information flows between such systems (Leyland, 1993). Information flows are used to exchange reference data, such as product information. They are also used to conduct business transactions between parties, such as the control of the flow of goods or services. In the latter case EDI is used to implement the information flows that control the primary flow of work or goods.

Information is exchanged in the form of messages. To conclude a business transaction, such as the ordering of transport, usually a number of messages have to be exchanged.

A. Sölvberg et al. (eds.), *Information Systems Development for Decentralized Organizations*
© Springer Science+Business Media Dordrecht 1995

These messages are related to one another. Their order and significance with respect to the business transaction is expressed in a business transaction protocol (Hofman, 1994).

The key issue of this effort is to make sure that the parties involved have a consistent view of the information exchanged.

Business transactions have been conducted long before the concept was introduced in database systems for describing a unit of work against the database (Gray and Reuter, 1993). In database systems various applications may concurrently need to access the database. In order to guard the data from becoming inconsistent concurrent execution of transactions has to be controlled (Bernstein et al, 1987). Data modified by a transaction is not released until that transaction has successfully completed. Other transactions that need to access that data have to wait in some schemes of concurrency control or are allowed to go ahead at the risk of having to restart at a later stage in some other schemes.

When a database is distributed over a number of sites, concurrency control has to take this into account. A transaction may need to access data at various sites. Work for the transaction then is performed at various sites and the sites have to agree that work everywhere is done correctly and consistently. Distributed commit protocols are enacted to make sure of this.

Database transactions are controlled by transaction managers, that keep track of the progress of the work at various sites and initiates the various steps in the transaction protocol. Since database systems are meant to achieve a large throughput of work, the transaction manager is an automated part of the system.

Business transactions are becoming in some respects comparable to database transactions. When business relations are stable the larger part of a transaction may be routine and proceed without interruptions by human operators, possibly subject only to a final approval. In addition, because of the automatic support, information exchanges will no longer be suspended until some point in time when the transaction has gained some critical mass, but they will become smaller in volume and proceed at the earliest time possible in order to gain certainty about the feasibility of the transaction as soon as possible. The number of transactions and therefore also the number of messages will increase.

A further complication is that transactions may become nested: in order to proceed with a transaction the support of third parties (subcontractors) may be needed. This situation occurs quite frequently when the two parties involved are part of a logistic chain. One party, that has a freight to be transported may contact a transporter, who will accept the job. This transporter may choose to do part of the work himself and have other transporters do other parts for him. Of course, he can only commit himself after he has obtained the commitment of the other transporters. At this point it may become a problem to keep the exchanges as well as their mutual dependencies manageable.

A solution for this problem can be achieved by automation of the control of the message exchange. A software utility is needed that monitors and controls routine message flows. In this paper the software solution takes the form of a generic transaction protocol handler. The transaction protocol handler maintains the correctness of the transaction as specified in the protocol. A transaction can be regarded as a collection of messages that are mutually related. The exchange of these messages is subject to a number of rules regarding order and content of the messages.

The transaction protocol handler will coordinate the communication (exchange of mes-

sages) with the other parties (actors) in the communication network, who may have transaction handlers of their own. It will constitute the interface of a party with the external world. Internally, the transaction protocol handler interfaces with a business process providing the physical activities (transportation in our example).

For a prototype to work, this aspect of the system has to be accounted for. In our prototype we therefore include a component representing a business application process. The application process serves as a model for the relevant parts of the capacity planning of the system. It decides on providing activities to external parties when requested. It generates messages on the basis of incoming messages and local information. In other words, it acts as a terminator for the network. The combination of a transaction protocol handler and an application process, which we will call message handler for brevity, models the communication behavior of a party in a communication network.

In this paper we discuss the design and implementation of a prototype for a message handler. In particular we will focus on the generic part of the prototype: the transaction protocol handler. The business application has been instantiated for the case of a logistical chain. Such a chain can be regarded as a decentralized system of autonomous units. It may involve parties ordering transport, brokers arranging transport, and carriers executing transport. In addition, also warehouses and distribution centers may be involved for temporary storage and distribution of the goods to be transported. The prototype therefore simulates the information exchange between parties in a transport chain. Alternate instantiations, e.g., internal logistics in a hospital, using HL7-messages, follow the same pattern. In this example, patients have to be admitted to the hospital, transferred between wards, laboratory tests have to be ordered and surgery has to be planned and carried out.

The prototype has been implemented in ExSpect(van Hee et al, 1989), a tool for writing executable specifications, which provides extensive support for performing and analyzing simulations of systems. This paper is based on the work described in (Koop, 1993). In the remainder of this paper we will focus on the architecture of the transaction protocol handler.

2 GLOBAL SOLUTION

The solution for efficiently dealing with the increase in the number of messages is sought in the form of a general utility for controlling the message flows. This utility has to satisfy a number of requirements following from the fact that it has to be able to recognize the state of a particular transaction and from the fact that one transaction may spawn off a number of secondary transactions. This is discussed in the following section. In the section after that we will show how the requirements lead to a general architecture for the message handler. In the final section some features of the implementation will be discussed. The solution proposed here is quite general and does not depend on the example chosen for the purpose of illustration: the control of EDI-message exchanges in a transportation chain. In order to emphasize this the following discussion will refer to *actors* instead of *parties.*

2.1 Requirements

The usefulness of a generic transaction protocol handler derives from the fact that it is able to keep track of the progress that has been made with a given business transaction, and of the relations between various business transactions. In order to understand the architecture it is necessary to have some appreciation of the complications arising in a business transaction and the difficulties a transaction protocol has to deal with. It is not our aim to discuss the design of protocols here.

Business Transactions

A business transaction, in the context of this paper, is regarded as the information exchange between two actors, needed to control a single physical activity, such as the transportation of a given number of goods from one location to another. The information is exchanged between two actors (parties): a *superior* who initiates the transaction by requesting the activity and a *subordinate*, who is asked to carry out the activity.

A business transaction has to satisfy a number of conditions, which resemble those for database transactions. An important difference here is that transactions control physical activities:

- An activity has to be carried out either completely or not at all. Partial completion of an activity is not allowed.
- The information about a given activity as stored by the superior has to be identical to that stored by the subordinate.
- To the superior it must appear as if the activity of a subordinate is executed independently of other activities executed by the same subordinate. This implies that the bound resources of these transactions, such as the means of transportation and the goods to be transported, have to be different.
- Once an activity has been completed no more messages may be exchanged concerning that particular activity. The result of an activity may only be changed by starting a new one, e.g., to compensate the effect of the activity.

These conditions appear to be the usual properties required for database transactions. With business transactions, however, some differences occur. First of all, parties are autonomous. Business transaction protocols there for are agreements that may be fixed between two business partners but may vary with different partners. Communication therefore is always between two parties. A transaction may involve more than one subordinate, but subordinates don't communicate with each other within the context of the transactions.

Local autonomy, that is the autonomy of each party implies, that in general no atomicity can be guaranteed (Mullen et al, 1991). There is no global transaction manager and only bilateral agreements can be used.

The information exchange is split up into an exchange of (standardized) messages. In order to preserve the properties mentioned above, a transaction is characterized by a special attribute: its state. In each state only certain messages can be meaningfully sent

and received. What these states are and which messages may be exchanged is specified in a protocol and represented in a state transition diagram, such as shown in Fig. 1. The labels used in this figure will be explained below. Fig. 1 is just one of many possibilities.

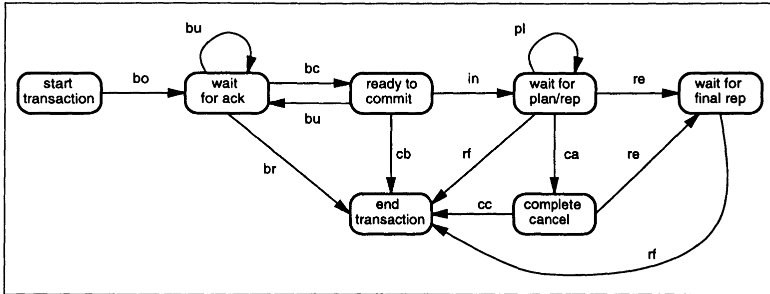

Figure 1: State transition diagram for a transaction.

The exchange of a message may change the state of the transaction as decisions with regard to certain aspects of the requested activity are communicated. In Fig. 1 an example of the various states of a transaction are given. Other possibilities are discussed in (Hofman, 1994). The arrows indicate the transition of one state to another (possibly the same) state, and are labeled by the type of message exchanged in making the transition. We will illustrate the diagram for the case of external logistics, but it is applicable to other situations (e.g., distribution) as well. Similarly, alternate state transition diagrams, involving more or different transitions and messages may be devised to reflect more properly the protocol needed.

The state transition diagram summarizes the protocol by specifying in which state which type of messages may be sent or received. The information exchange pass through three phases. In the obligatory **prepare** phase of the transaction the resources needed for a certain activity are reserved. The superior sends (from the initial state) a *booking* (bo-message). When the subordinate wants to carry out the activity and has sufficient resources to do so, he replies with a *booking confirmation* (bc-message). The resources needed will be reserved and become 'bound' resources. This compares to the locking mechanism used to implement database transactions. In the other case a *booking reject* is sent (br-message). By means of a *booking update* (bu-message) the superior may communicate modifications or additional information. The activity may not be changed, however. (When a different activity is required, the current transaction has to be canceled and a new one has to be started.) The superior may call off a transaction in this phase by sending a *cancel booking* (cb-message).

The **commit** phase begins as soon as the superior sends an *instruction* (in-message). The subordinate keeps the superior informed of the schedule of the required activity by sending a *planning* (pl-message). Information about the actual progress of the activity is transmitted by means of a *report* (re-message). When the activity is completed a *final report* (rf-message) is sent by the subordinate and the transaction ends.

When the superior decides in the commit phase to call off the activity, he sends a

cancel (ca-message). The transaction now enters the **abort** phase. When the subordinate has already started the activity, he may reject the cancellation by sending a *report* (re-message). If not, he sends a *cancel confirmation* (cc-message) and the transaction ends. Note that the protocol has to exclude concurrency between various subordinates, in order to avoid conflicts between some subordinates having started the (physical) activity and others still waiting for the confirmation.

Relationships between transactions

The role of superior or subordinate is only fixed within the context of a single transaction. The dual role of an actor with respect to transactions in general is illustrated in Fig. 2.

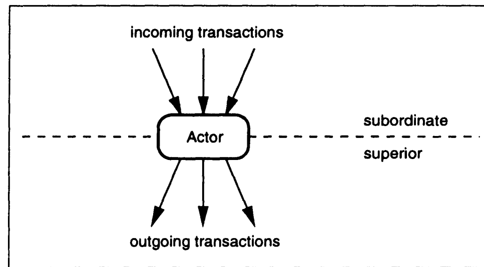

Figure 2: Incoming and outgoing transactions.

When we are dealing with a chain of activities an actor may act both as a superior and as a subordinate (for different transactions). For instance, an actor, say Carrier_1, who is asked by another actor - Shipper_1, acting as a superior - to provide transportation services, acts as a subordinate. In case Carrier_1 doesn't have the capacity needed, he may initiate an outgoing transaction to ask a third actor, say Carrier_2, to provide the missing capacity (as a subcontractor) and then acts as a superior to Carrier_2. Note that the incoming transaction is dependent of the result of the secondary, outgoing transaction: Carrier_1 can't confirm the booking of Shipper_1 before the missing capacity has been reserved elsewhere. Such an outgoing transaction is called an *inserted* transaction (see Fig. 3). It satisfies the following two conditions:

- A message of the outgoing transaction can only be sent after a message of the same type of the incoming transaction has been received.
- A message of the incoming transaction can only be sent after a message of the same type of the outgoing transaction has been received.

For instance, an *instruction* message of an inserted transaction can only be sent after an *instruction* message of the corresponding incoming transaction has been received. Every phase has to be initiated by Shipper_1, the superior of the initial transaction.

In case an incoming transaction causes an outgoing transaction but doesn't depend on the result of it, the outgoing transaction is called a *decoupled* transaction (see Fig. 4). E.g., a supplier may be able to fill an order for a certain quantity of goods (a booking

message), but may be obliged to put out an order to a producer for the same type of goods, because the stock level has fallen below some limit. Clearly, the completion of the first transaction doesn't have to wait for the progress of the second one. The store of the supplier acts as a decoupling point.

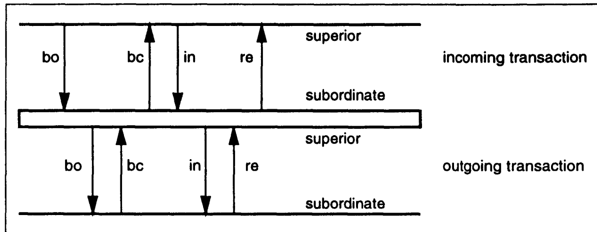

Figure 3: Inserted Transaction.

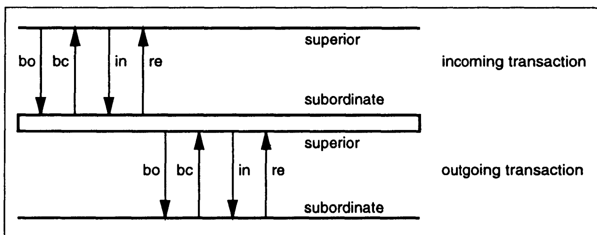

Figure 4: Decoupled Transaction.

2.2 Architecture

The requirements of the preceding section may be summarized as follows:

- We have to distinguish between incoming and outgoing messages. Depending on the role of an actor and the state of the transaction, a message of a certain type may be sent or received.
- We have to distinguish between incoming and outgoing transactions. An actor may act as a superior or as a subordinate. In the former case a transaction is called outgoing, in the latter incoming.
- We have to keep track of the relationship between transactions. An outgoing transaction may be inserted into an incoming transaction or it may decoupled from it. In the former case the incoming and outgoing transaction have to be synchronized.

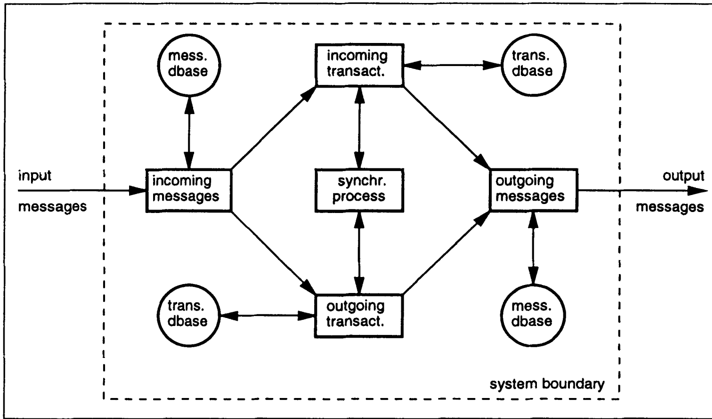

Figure 5: System architecture.

The requirements above lead to the architecture, shown in Fig. 5 (subsystems, i.e. processes, represented by squares, stores by circles, flows by arrows).

3 EXSPECT

ExSpect (Executable Specification tool) is a formalism to describe distributed systems (van Hee et al, 1989). It is based on timed hierarchical colored Petri Nets. (For colored Petri Nets, see e.g. (Jensen, 1992).)

Basically, the description of a system in the ExSpect formalism consists of two kinds of objects: channels (places) and processors (transitions). A channel is passive, it serves as a place holder of so called tokens: at any moment in time it contains a number of these tokens. A channel can serve as input or output channel for a processor, see for example Fig. 6.

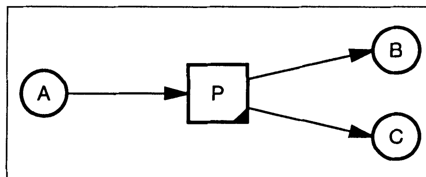

Figure 6: A processor (P) with input channel A and output channels B and C.

A processor is active: if all input channels of a processor contain at least one token, the processor may fire. If a processor fires, it removes (consumes) a token from each of its input channels and produces tokens for its output channels.

Each channel has a type that specifies which values (colors) are allowed for the tokens in that channel. The values of the tokens that are produced when a processor fires depend upon the values of the tokens that are consumed. Thus, a processor can be seen as a function from the types of its input channels to the types of its output channels.

A processor may have a precondition: only tokens with values that obey this precondition will be consumed.

Tokens also carry a time stamp. A token may not be consumed before the (simulated) time is greater than or equal to its time stamp. In the processor a delay is specified for each produced token. The time stamp is equal to the time of firing plus the delay.

```
proc P [in A: num, out B: num×str, C: str] :=
   if A > 0 then
      B ← ≪A+1,'ok'≫ delay 3.5
   else
      C ← 'invalid'
   fi
```

Figure 7: A possible definition of processor P.

The example in Fig. 7 represents a processor that consumes one numerical token from channel A at a time. If this token is positive then a token is produced for channel B. Its value is a pair consisting of the incremented value of the consumed token and the string 'ok'. The time stamp of this token will be equal to the time of firing plus 3.5. If the consumed token is not positive, a token is produced for channel C without delay.

The colors of the tokens may represent very complex data. The ExSpect type system has a few basic types and type constructors to build more complicated types. Among the basic types are the booleans (bool), rational numbers (num), floating point numbers (real) and strings (str). The type constructors allow for the construction of Cartesian products, sets, finite mappings (sets of pairs with unique first component) and tuples.

Models of large systems will become very soon too complex to understand if they consist of only channels and processors. The hierarchy concept solves this problem. Channels and processors may be grouped into subsystems ('processes'), such subsystems may be used in other subsystems and so on.

ExSpect has been implemented in a set of cooperating tools to design, simulate and analyze a specification. A graphical editor may be used to create the flow diagrams, a type checker checks the consistency and correctness of the specification and the simulator offers the possibility for interactive or batch simulation. During an interactive simulation the user may interact with the running simulation (inspecting places, adding tokens, reading from and writing to files or other applications) without stopping the simulation. Animation of token production and consumption is optional. Furthermore a static analysis tool to calculate some structural properties and timing properties is available.

The results of a simulation may be analyzed on-line or off-line by means of other software, like a spread sheet program. Alternatively, the simulation results may be used for conformance testing of a final application.

4 IMPLEMENTATION

When we analyze the architecture, proposed in Fig. 5, we can identify a number of system components. The resulting (ExSpect) process model is given in Fig. 8. In the diagram only the processors and the channels having to do with the processing of the messages have been shown. The database aspect has been suppressed, as well as a number of channels that can be attached for error handling and simulation purposes.

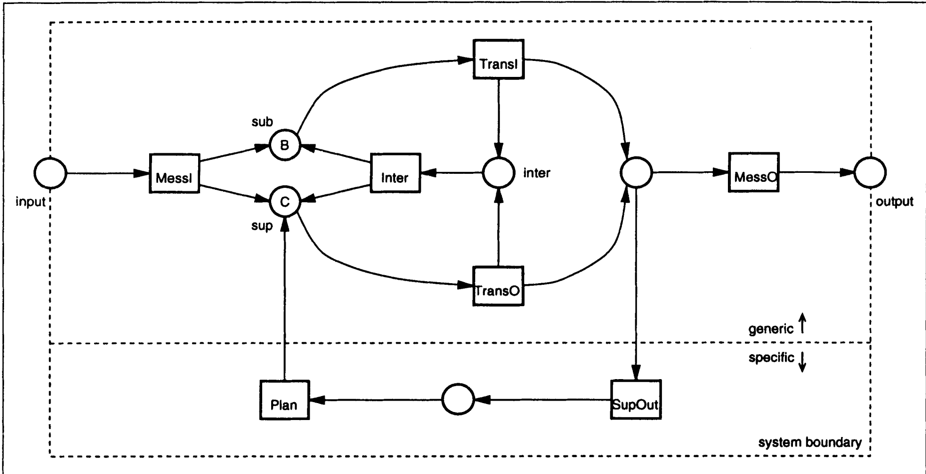

Figure 8: Process Model.

To arrive at the model of Fig. 8 starting from the architecture in Fig. 5 some observations need to be made. The heart of the transaction handler is formed by the processes 'TransI' and 'TransO'. 'TransI' and 'TransO' keep the state of incoming and outgoing transactions, respectively, up to date. Process 'Inter' keeps track of the relationships between incoming and outgoing transactions.

The generic transaction handler is formed by processes 'MessI/O', 'Inter' and 'TransI/O'. The application dependent part is limited to 'SupOut' and 'Plan'. So, if we want to use the message handler for a field other than external logistics, we only have to adapt these two processes.

When we treat requests for activities that are offered by the actor itself (i.e., for which it has sufficient resources in a given period) in the same manner as requests for which the cooperation of other parties is needed (i.e., the actor does not have sufficient resources itself or has none at all), the functionality of 'TransI' and 'TransO' becomes identical. This has as a consequence that for every bo-message entering the system from the outside an inserted outgoing transaction will be started.

Consider, e.g., what will happen when a bo-message is received. 'TransI' notes the start of a new transaction and passes the request on to 'Inter'. 'Inter' notes that an inserted transaction will be needed and generates the required bo-message for 'TransO'.

'TransO' notes then the start of an outgoing transaction. When the booking is destined for an external actor, it passes 'SupOut' and is processed by 'MessO' into an external message. When the bo-message is destined for internal processing, 'SupOut' filters it out and passes it to 'Plan', where a decision is taken: reject or confirm on the basis of the available capacity. A br- or bc-message is generated, respectively, and sent to 'TransO' which notes the change in state of the transaction. The message then is propagated further via 'Inter' and 'TransI' to 'MessO' from where it is sent to the superior.

We will now discuss each of the processes in more detail for the case of external logistics.

MessI

The messages enter the system via a single (complex) process. It is implemented as a single processor as can be seen in Fig. 9. In this figure channel 'err' and store 'message' are shown that had been suppressed in the main diagram of Fig. 8. Channel 'out1' is connected to 'sub' and 'out2' to 'sup'.

As can be seen from the specification in Fig. 10, the processor that handles the incoming messages (labeled 'messInfilter') checks to see if the message hasn't been previously received (not(MessPresent(kk,Message@MESS))), determines whether it is part of an incoming (mm $\in$ {'bo',...}) or outgoing transaction (mm $\in$ {'br',...}), and whether it has its logical place (i.e., the correct sequence number) for the transaction concerned (MessSucc(kk,Message@MESS)). If so, the message is stored (Message $\leftarrow$ ...) and converted to an internal representation, which is passed on (out1,2 $\leftarrow$ ...). If not, an error message is generated. The precondition (pre r = me) guarantees that only messages for this node are selected.

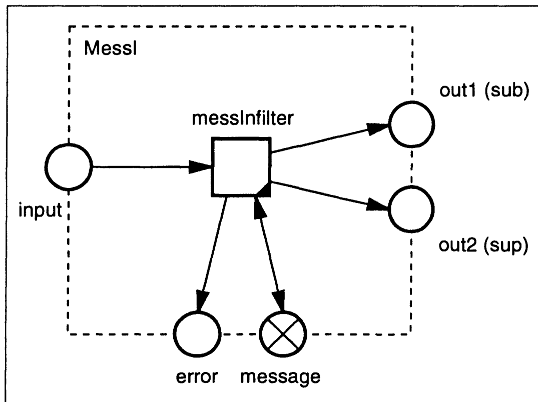

Figure 9: The handler of incoming messages (MessI).

TransI/TransO

Incoming messages of incoming transactions, i.e., messages of type 'bo', 'bu', 'cb', 'in' or 'ca' (see Fig. 1), which come from a superior and are called superior messages, are passed on to the process 'TransI'. Those of type 'br', 'bc', 'cc', 'pl', 're' or 'rf', which come from a

```
proc messInfilter
 [  in      input: Mess,
    out     out1: Token,
            out2: Token,
            error: Error,
    store   Message: MESSAGE,
    val     me: Id |
    pre     r = me
 ]
:=
   if not(MessPresent(kk,Message@MESS)) and MessSucc(kk,Message@MESS) then
      if mm ∈ {'bo','bu','cb','in','ca'} then
         Message ← upd(Message, [MESS:InsMessIN(kk,d,Message@MESS)]),
         out1 ← upd([k:p],d)
      elif mm ∈ {'br','bc','pl','re','rf','cc'} then
         Message ← upd(Message, [MESS:InsMessIN(kk,d,Message@MESS)]),
         out2 ← upd([k:b],d)
      else
         error ← 'MessIn error'
      fi
   else
      error ← 'MessIn error'
   fi
where
    kk  := input@k: MessKey;
    mm  := input@m: MessType;
    r   := kk@receiver: Id;
    s   := kk@sender: Id;
    t   := kk@trans: Id;
    d   := [m:input@m, a:input@a, l:input@l]: MessData;
    b   := [sub:s, sup:r, trans:t]: TransKey;
    p   := [sub:r, sup:s, trans:t]: TransKey
end
```

Figure 10: The definition of processor 'messInfilter'. Mess is a tuple type containing a.o. a field of type Messkey with label k and a field of type MessType with label m: type Mess := [k:MessKey, m:MessType, ...].

subordinate and are called subordinate messages, belong to an outgoing transaction and are passed on to the process 'TransO'.

The processes 'TransI' and 'TransO' are instances of a common process, 'Transaction'. The difference between these two instances lies in the fact that the role of the two output channels has been interchanged. Both processes fulfill the task of keeping the transaction data up to date. In addition the correctness of the data about the activities which are being required, planned or performed, is checked and in case of violations an error message is generated.

When the process 'TransI' is activated, the system is acting as a subordinate. The process will generate superior messages for the synchronization process 'Inter', and subordinate messages for the process 'MessO'.

When the process 'TransO' is activated, the system will act as a superior. The process will generate superior messages for the process 'SupOut' and subordinate messages for the process 'Inter'.

The process 'Transition' consists of a number of elementary subprocesses, one for each type of message. Each subprocess picks out only those messages from the input channel that belong to its message type. The subprocess then checks to see whether the message can be received in the current state of the transaction. It also checks whether the data are consistent with previously received messages.

E.g., consider the subprocess 'TransBu', which handles booking updates and processes exclusively messages of type 'bu'. TransBu checks to see whether the current state of the transaction is "wait for ack" or "ready to commit", which, according to Fig. 1 are the only states in which a booking update message may be received. Then it checks to see whether the data have acceptable values (i.e., belong to the right type) and whether the required activity is the same as the one in the original booking for the same transaction. If all checks out to be correct, the booking is updated (e.g., a different date or time is set) and the state of the transaction is changed to 'wait for ack'. The message is sent to the process 'Inter'.

Inter

The process 'Inter' takes care of the synchronization of the incoming and the outgoing transactions. It is at the core of the message handler. For this purpose it records status information about the relationships between transactions. 'Inter' initiates inserted transactions and generates the proper messages for 'TransI' and 'TransO' in response to messages (information) it receives from 'TransO' and 'TransI' respectively.

The process has one input channel, which receives messages of both incoming (via TransI) and outgoing (via TransO) transactions. It can send messages to either the superior or the subordinate system.

Again the process is split up into a number of subprocesses, one for each message type. In this case the subprocess handling the booking messages has a special structure: it has an extra input channel. There it can receive not only the bo-messages of a new incoming transaction but also the confirmation and rejection of an outgoing booking. This may cause the start of a new outgoing transaction, in the former case because certain (supporting) activities should only be requested once the main activity is confirmed. In the latter case, because a rejection may necessitate the request of an activity from an

alternate supplier.

E.g., a booking update (bu) message from TransI will nullify the number of acknowledged outgoing transactions. For every outgoing transaction a new bu-message is composed and passed to the superior output channel.

SupOut

An actor in the role of a subordinate has the choice between performing a requested activity itself and asking a third actor to carry out the activity, when its own capacity appears to be insufficient. When we introduce a special subordinate actor, representing the internal capacity, the process Inter can be specified at a completely general level, without the need to make a distinction between internal and external processing. It just starts up transactions to potential suppliers of activities, including itself. The messages requesting activities from external suppliers are passed to 'MessO'. The other messages requesting internal activities do not need to leave the system but are, instead, filtered out by 'SupOut' and sent to the 'Plan' process.

MessO

This process translates internal messages to external ones. It checks to see if the superior identification of messages of type 'bo', 'bu', 'cb', 'in' and 'ca' and the subordinate identification of messages of the type 'br', 'bc', 'cc', 'pl', 're' and 'rf' is the identification of the actor itself. If so, the message is translated, stored in the message database and sent out. If not, an error is produced.

Plan(ning)

This process models the interface between the transaction handler and the application. It simulates the capacity planning aspects of the problem. The process has three major subprocesses: 'Reserve', 'Release' and 'Monitor'.

When a booking message is received, it is passed to 'Reserve'. There the available capacity is checked and a decision is taken. In case of sufficient capacity, i.e., sufficient resources that haven't been bound yet in the period at hand, the requested resources are bound for the period and a confirmation (bc-message) is sent to the superior component. If not, a br-message is generated.

Messages of type 'bu', 'cb' and 'ca' are passed to 'Release' and cause bound resources to be released. The 'bu' message then is passed on to 'Reserve' to see if the new requirements can be met. In response to a 'ca' message a confirmation is generated (cc-message). In case of a 'cb' message no action is needed (in this simulation).

The 'Monitor' checks for activities that should start. When the transaction is still in the preparatory phase, a 'br'-message is sent. In the commit-stage a 're'-message is sent. Finally when an activity is about to end an 'rf'-message is generated, in case the transaction is in the commit-phase.

The response to an 'in'-message is to generate a forecast (pl-message) of the expected activities.

5 CONCLUSIONS

In this paper an ExSpect prototype of an actor in a logistic chain has been described. The prototype contains generic software for supporting Electronic Data Interchange. The prototype implements a utility for processing data and controlling various primary business processes.

The utility has been illustrated for the case of external logistics but we could have equally well taken our case from the storage and distribution areas. The instantiation of the solution to another application area will not have any consequences for the systems architecture. The architecture is based on the processing of data by means of transactions, and is independent of the content of the messages used.

The system offers a flexible implementation of protocols. We have demonstrated that not only simple protocols, but also the weaving of protocols can be handled in a flexible way. This flexibility stems from the architecture of the system: dedicated subprocesses for each message type and clear modules for handling the control of single transactions and the interplay between transactions respectively.

The use of an executable specification language for designing the system offers the advantage of being able to simulate the message exchange between a number of parties, each of which is an instantiation of the message handler system with its own set of resources and activities offered. By simulating the message exchange for a number of cases, the protocols can be validated. It then becomes possible to design tailor made protocols which are optimally suited for the control of message exchanges between specific parties. Having a flexible system then pays off: the impact of adding a message or transition is quite local and its consequences are easy to grasp and validate.

The prototype supports the modeling and analysis of various designs for a decentralized organization. Various assignments of tasks to locations (system configurations) can be proposed and subsequently analyzed in terms of the complexity of the communication needed to control the flow of work or goods between these locations. This should yield useful insight in the viability of the various designs.

6 REFERENCES

Bernstein, P.A., Hadzilacos, V., Goodman, N. (1987) *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Co.

Gray, J. and Reuter, A. (1993) *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc.

van Hee, K.M., Somers L.J., Voorhoeve, M. (1989) Executable specifications for distributed information systems, in *Information system concepts: an in-depth analysis,* (eds. E.D. Falkenberg, P. Lindgreen), North-Holland.

Hofman, W.J. (1994) *A conceptual model of a Business Transaction Management System.* Tutein Nolthenius, Amsterdam.

Jensen, K. (1992) *Coloured Petri nets. Basic concepts, analysis methods and practical use.* Vol. 1: Basic concepts. EATCS monographs on Theoretical Comp. Sci., Springer-Verlag.

Koop, C. (1993) *ExSpect prototype of a party in External Logistics.* Masters thesis, Eindhoven University of Technology. (In Dutch.)

Leyland, V. (1993) *Electronic Data Interchange.* Prentice Hall International.

Mullen, J.G., Elmagarmid, A.K., Kim, W., Sharif-Askary, J. (1991) On the impossibility of Atomic Commitment in Multidatabase Systems, in *Proc. First Int'l Workshop on Interoperability in Multidatabase Systems*, pp. 625-634.

7 BIOGRAPHY

Ad Aerts received his Ph.D. from the University of Nijmegen in 1979. He has held several postdoc positions in the U.S. and Europe prior to his current position as lecturer at the Eindhoven University of Technology. He has published a large amount of papers on various subjects at conferences and scientific journals. His research interests currently include database models, integration of database systems and transaction management.

Wout Hofman is senior consultant in the area of telematics with Bakkenist Management Consultants. He holds a masters degree in electrical engineering and information processing. The subject of his Ph.D. thesis (1994, Eindhoven University of Technology) is the conceptual model of a Business Transaction Management System based on EDI communication. He has published a number of papers on the subject of EDI and is the author of the Dutch EDI handbook. He is also a member of the advisory board of the ministry of transport and travel with respect to telematics.

Lou Somers received his Ph.D. in theoretical physics from the University of Nijmegen in 1984. For several years he worked as a systems engineer at Philips Data Systems, a.o. on the design and development of a system for distributed databases and a user interface management system. Currently he is associate professor in the area of software engineering at the Eindhoven University of technology. He has designed the software package ExSpect. Furthermore, he has been involved in several national Dutch projects, the ESPRIT project PROOFS, and the TEDIS project EDISCAN.