On the Duality of Fault Tolerant System Structures

(Preliminary Version)

S.K. Shrivastava, L.V. Mancini and B. Randell
Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, U.K.

ABSTRACT
An examination of the structure of fault tolerant systems incorporating error recovery, and in particular backward error recovery, indicates a partitioning into two broad classes. Two canonical models, each representing a particular class of systems have been constructed. The first model incorporates objects and actions as the entities for program construction while the second model employs communicating processes. Applications in the areas such as office information and database systems typically use the first model while applications in the area of real time process control are usually based on the second model. The paper claims that the two models are duals of each other and presents arguments and examples to substantiate this claim, which is in effect, an extension of the earlier duality argument presented by Lauer and Needham. An interesting conclusion to be drawn from this study is that there is no inherent reason for selecting one model over the other, but that the choice is governed by the architectural features of the layer over which the system is to be constructed. A pleasing consequence has been the recognition that the techniques which have been developed for one model, turn out to have interesting and hitherto unexplored duals in the other model.

Index Terms: fault tolerance, reliability, distributed systems, object based systems, real time systems, operating systems.

1. Introduction

An investigation of backward error recovery based fault tolerance techniques employed in a variety of systems reveals two general classifications. We propose two models, each embodying the major characteristics of the corresponding class of systems. One widely used technique of introducing fault tolerance - particularly in distributed systems - is based on the use of *atomic actions* (atomic transactions) for structuring programs [1]. An atomic action possesses the properties of serializability, failure atomicity and permanence of effect. Atomic actions operate on *objects* (instances of abstract data types). The class of applications where such an *object and action* (OA) based model has found usage include banking, office information, airline reservation and database systems. A number of other applications - typically concerned with real time control - are structured as concurrent processes communicating via messages. Some examples are process control, avionics and telephone switching systems. Fault tolerance in such systems is introduced through a controlled use of *checkpoints* by processes. We will refer to this way of structuring an application as employing the *process and message* (PM) model.

In this paper we claim that the OA and PM approaches to the provision of fault tolerance are duals of each other and present arguments and examples to substantiate our claim. As a result of this observation, we can state that there is no *inherent* reason for favouring one approach over the other; rather the choice is largely dictated by the architectural features of the underlying layer. Indeed, we would now claim that the differences between the two approaches are basically a matter of viewpoint and terminology. Our investigations have been influenced by the well known duality paper of Lauer and Needham [2] which puts forward the notion that within the context of operating systems, procedure based systems and message based systems are duals of each other. The authors observed that (1) a program or subsystem constructed strictly according to the primitives defined by one model can be mapped directly into a dual program or subsystem which fits the other model; (2) the dual programs or subsystems are logically identical to each other, and they can also be made textually very similar; and (3) the performance of a program or subsystem from one model will be identical to its counterpart. The present work may be considered as an extension of the ideas put forward in that paper with regard to fault tolerance.

The paper is structured as follows: sections two and three describe the essential aspects of OA and PM models respectively. Section four contains the arguments intended to establish the duality between OA and PM. Section five contains a few simple examples, and the concluding section summarizes the paper and discusses possible implications of the duality claim. Throughout the paper, we will assume a distributed system composed out of a number of nodes connected by some communication medium.

2. Object and Action Model

Objects are instances of abstract data types. An object encapsulates some data and provides a set of operations for manipulating the data, these operations being the only means of object manipulation. In most object based fault tolerant systems that we know (see [3-8] for a representative sample), an operation is performed by invoking an object with a *remote procedure call* (RPC), which passes value parameters to the object and returns the results of the operation to the caller. Programs which operate on objects are executed as *atomic actions* with the properties of (i) *serializability*, (ii) *failure atomicity* and (iii) *permanence of effect* [1]. The first property ensures that concurrent executions of programs are free from interference (i.e. a concurrent execution can be shown to be equivalent to some serial order of execution [9,10]). The second property ensures that a computation can either be terminated normally, producing the intended results or be *aborted*, producing no results. This property is obtained by appropriate use of backward error recovery, which is invoked whenever a failure that cannot be masked occurs. Typical failures causing an action to be aborted are node crashes and communication failures such as lost messages. It is reasonable to assume that once a computation terminates normally, the results produced are not destroyed by subsequent node crashes. This is the third property - permanence of effect - which ensures that state changes produced are recorded on *stable storage* which can survive node crashes with a high probability of success. A *two-phase commit* protocol is required during the termination of an action to ensure that either all the objects updated within the action have their new states recorded on stable storage (normal termination), or no updates get recorded (aborted termination).

A variety of concurrency control techniques for atomic actions to enforce the serializability property have been reported in the literature. A very simple and widely used approach is to regard all operations on objects to be of type *read* or *write*, which must follow the well known locking rule permitting *concurrent reads but only exclusive writes*. In a classic paper [9], Eswaren *et al.* proved that actions must follow a *two-phase* locking policy (see Fig. 1). During the first phase, termed the *growing phase*, a computation can acquire locks on objects but not release them. The tail end of the computation constitutes the *shrinking phase*, during which time held locks can be released but no locks can be acquired. Now suppose that an action in its shrinking phase is to be aborted, and that some updated objects have been released. If some of these objects have been locked by other actions, then abortion of the action will require these actions to be aborted as well. To avoid this *cascade abort* problem, it is necessary to make the shrinking phase *instantaneous*, as indicated by the dotted lines.

Any atomic action can be viewed at a lower level as constructed out of more primitive atomic actions - this is illustrated in Fig. 2 which also introduces the action diagram which will be used in this paper (this notation is based on that used by Davies [11]). According to Fig. 2, action B's constituents are actions B_1, B_2, B_3 and B_4. A directed arc from an action (e.g. A) to some other

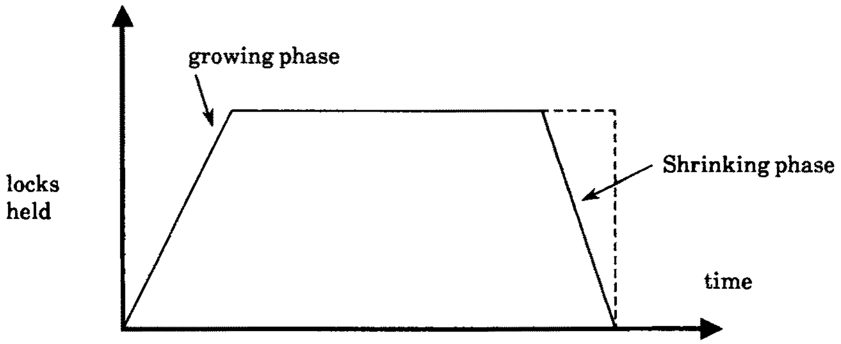

Fig. 1. Two phase locking.

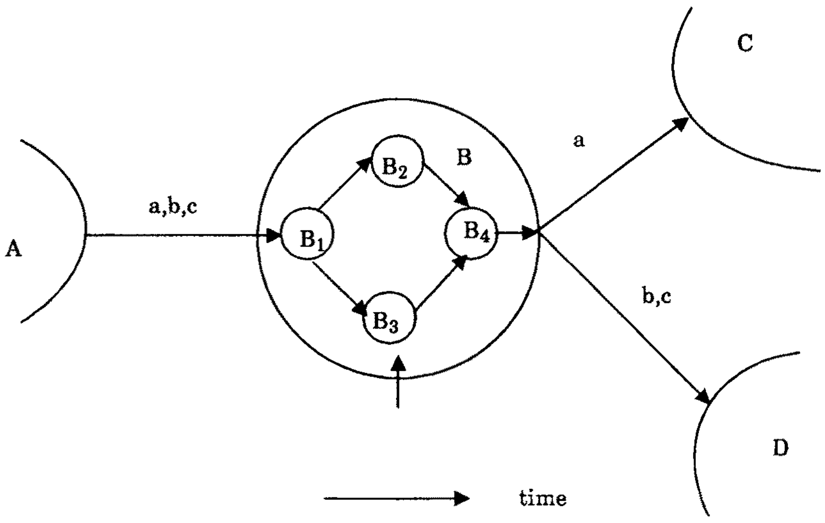

Fig. 2. Action diagram.

action (e.g. B) indicates that B uses objects released by A. Optionally, an arc can be labelled, naming the objects used by the action. In Fig. 2, B uses objects *a, b* and *c* and C uses object *a* which has been released by B. Actions such as B_2 and B_3 are executed concurrently. Nested actions give rise to nested recovery. Suppose time has advanced up to the point shown by the vertical arrow, and an error is detected in B_3 causing it to be aborted. What happens after B_3's recovery? The question must be resolved within the scope of B - the enclosing action. B can provide a specific *exception handler* to deal with this particular eventuality (such exception handling techniques have been discussed by Taylor [12]). If no handler is available, then a failure of B_3 will cause B to be aborted.

One of the most important aspects of the OA model from our point of view is the fact that objects and actions are the two primary entities from which an application program is constructed. Any implementation of actions and objects will require processes (clients and servers) for carrying out the required functions. However, the role played by processes is *hidden* at the application level. Similarly, there is no explicit use of message passing between entities, since RPCs hide the details of message interactions between clients and servers. For example, in the Argus programming system [3], the implementation of *guardians* (objects) requires a number of processes for receiving and executing calls from clients - but processes are not visible entities to be used explicitly by an application program. Taylor [12] describes a number of ways of implementing atomic actions using different process structures. In the OA model, objects are *long lived* entities and are the main repositories for holding system states, while actions are *short lived* entities.

3. Process and Message Model

In contrast to the OA model, where processes and messages play at most a secondary role, the PM model has them as the primary entities for structuring programs. An application is structured out of a number of concurrent and interacting processes. A notation for describing the PM model that has received much attention is the communicating sequential processes (CSP) notation [13] which can be used for specifying a concurrent system by a fixed number of processes interacting via synchronous message passing. The topic of backward error recovery among interacting processes has been studied extensively, e.g.[14-18], beginning with the study reported in [19].

The PM model will be assumed to have the following characteristics: (1) processes do not share memory, at least explicitly, and communicate via messages sent over the underlying communication medium; (2) appropriate communication protocols ensure that processes can send messages reliably such that they reach their intended destinations uncorrupted and in the sent order; (3) a process can take a *checkpoint* to save its current state on some reliable storage medium (stable storage). If a process fails, it is rolled back to its latest checkpoint.

The notion of a *consistent global state* of a system is central when considering the recovery of interacting processes. A global state of a system is the set of local states, one from each process (a precise formulation is presented in [20]). The interactions among processes can be depicted using a time diagram, such as that shown in Fig. 3. Here, horizontal lines are time axes of processes and sloping arrows represent messages. A global state is a cut dividing the time diagram into two halves. A cut in the time diagram is *consistent* (consistent global state) if no arrow starts on the right hand side of the cut and ends on the left hand side of it. Cut C_1 in the figure is consistent; but cut C_2 is not, since it indicates that process q has received a message which has not yet been sent by r.

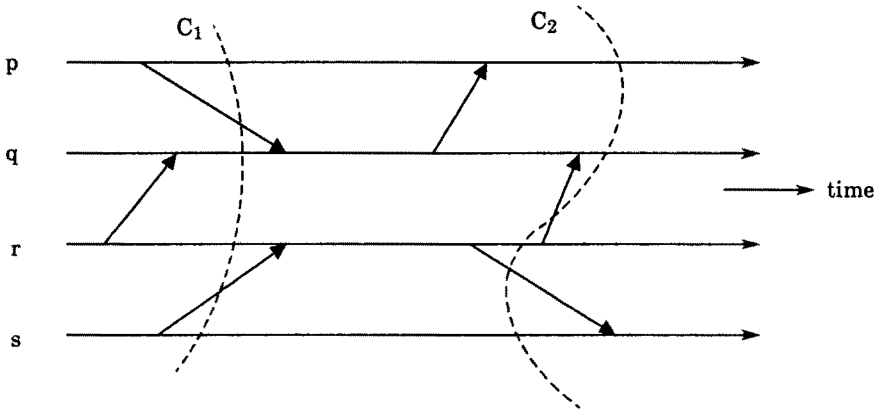

Fig. 3. Consistent and inconsistent cuts.

In a system of interacting processes, the recovery of one process to its checkpoint can create an inconsistent global state, unless some other relevant processes are rolled back as well. This leads to the notion of a *consistent set of checkpoints* or a *recovery line* [21]: a set of checkpoints, one from each process, is consistent if the saved states form a consistent global state. Fig. 4 illustrates the notions of consistent and inconsistent sets of checkpoints where opening square brackets on

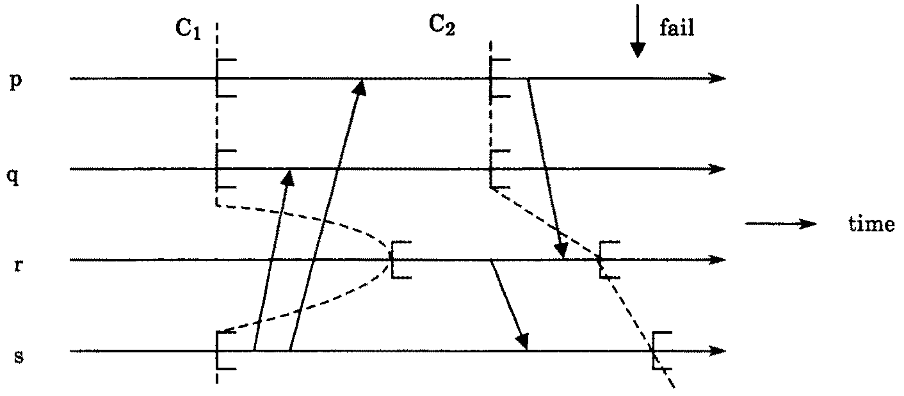

Fig . 4. Consistent and inconsistent sets of checkpoints.

process axes indicate checkpoints. Suppose process p fails at the point indicated by the vertical arrow and is rolled back to its latest checkpoint. The global state of the system as represented by cut C_2 is clearly inconsistent; the set of checkpoints on recovery line C_1 is however consistent. Thus a failure of p can cause a cascade rollback of all the four processes - this is the *domino effect* mentioned in [19]. The dynamic determination of a recovery line is a surprisingly hard task; the reader should consult [17] for a clear exposition.

The domino effect can be avoided if processes coordinate the checkpointing of their states. A well known scheme of coordinated checkpoints is the *conversation* scheme [15,19]. The set of processes which participate in a conversation may communicate freely between each other but with no other processes. Processes may enter the conversation at different times but, on entry,

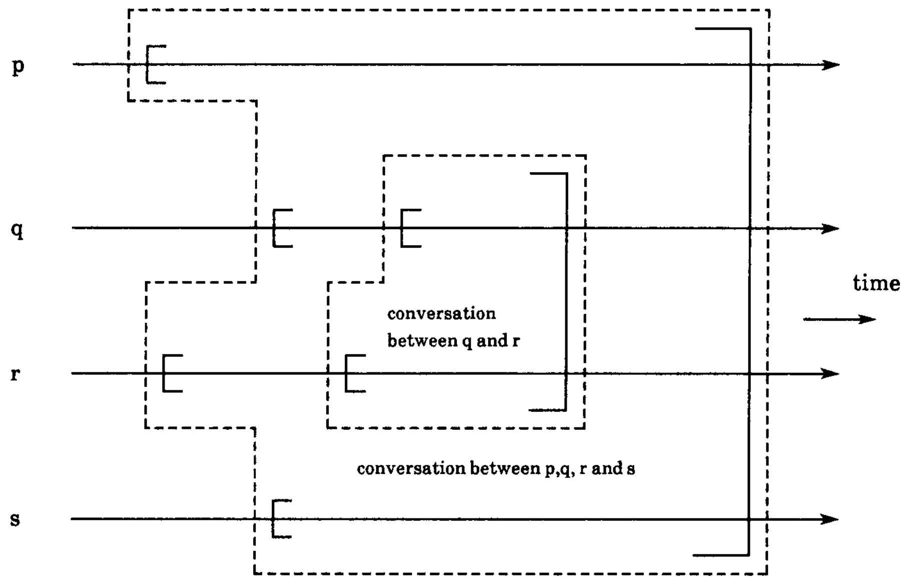
p
q
r
s
time
conversation between q and r
conversation between p,q, r and s

Fig . 5. Conversations.

each must establish a checkpoint (see Fig. 5). In Fig. 5, a closing bracket indicates that all participating processes must exit at the same time (brackets will not be explicitly drawn in the subsequent diagrams). If a process within a conversation fails then all the participating processes are rolled back to the respective checkpoints established at the start of the conversation. Conversations can be nested as shown in the figure.

Conversations provide a convenient structuring concept for introducing fault tolerance in a large class of real time systems [22]. The need to respond promptly to changes in the external environment dictates that most real time systems have an iterative nature. The PM model provides a natural way of expressing such systems in the form of interacting cyclic processes with synchronization points usually associated with timing constraints. A study of real time system structure for avionic systems by Anderson and Knight [22] indicated that synchronization of processes in such a system stems from the need to synchronize with the events in the external environment, rather than from any inherent needs of processes themselves. Fig. 6 depicts a typical synchronization requirement. An informal interpretation of such a *synchronization graph*

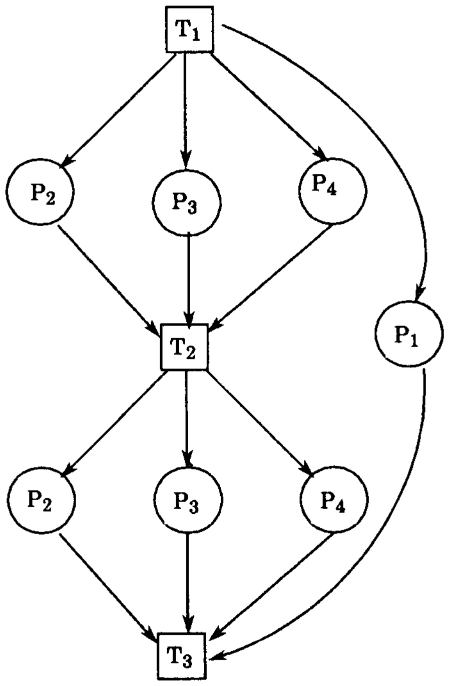

Fig. 6. A Synchronization graph.

is as follows (see [22] for a precise formulation): process P_1 repeatedly initiates a computation at time T_1 which must finish by time T_3 ($T_3 > T_1$); processes P_2, P_3 and P_4 complete two iterations in the interval T_1 to T_3. Any interactions between P_2, P_3 and P_4 can be performed within the confines of two conversations: one starting at T_1 and finishing at T_2 and the other starting at T_2 and finishing at T_3. The use of conversations for introducing fault tolerance in the manner indicated here is discussed at length in [22].

The most important aspects of the PM model relevant to this paper are summarized below. An application is programmed in terms of a number of processes interacting via message passing. If processes establish checkpoints in an arbitrary manner then there can be a danger of cascade rollback, which is usually undesirable. Conversations provide a coordinated means of managing checkpoints to avoid the danger of such a cascade rollback. However, a conversation requires the participating processes to synchronize such that they exit from the conversation simultaneously. A large class of applications, typically concerned with process control or real time control, traditionally employs the PM model for structuring applications. Conversations can be imposed on such applications by exploiting naturally occurring synchronization points among interacting processes. In the PM model, processes are *long lived* entities and main repositories for holding system states, while conversations are *short lived* entities.

4. Duality

The canonical models discussed in the previous two sections are representative of the corresponding class of fault tolerant systems. Given a description of any fault tolerant system, it is usually straightforward to work out its representative model, despite the fact that the terminology used for the description may even differ some what from that used here. The duality between the OA and PM models can be established by considering objects and actions to be the duals of processes and conversations respectively. Further, RPCs can be considered duals of messages [2]. A given conversation diagram (e.g. Fig. 7.a), can be translated into an action diagram quite simply (e.g. Fig. 7.b) by replacing each conversation C_i with a corresponding action A_i, and adding an arrow from A_i to A_j if C_i and C_j have at least one process in common and that process enters C_j after exiting from C_i. An arc from one action to the other is labelled with the

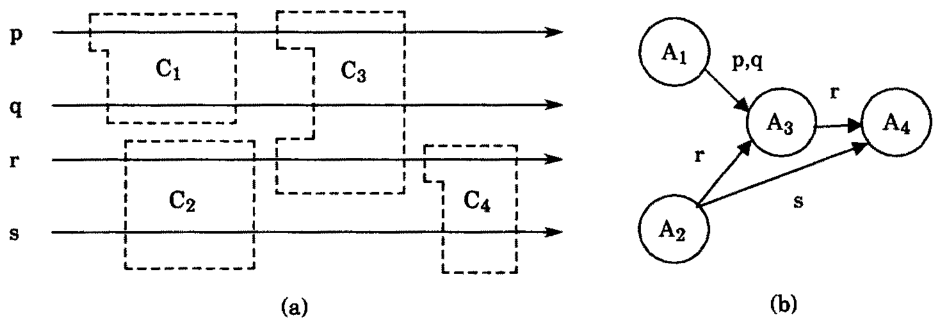

Fig . 7. Conversations and Actions.

objects representing the processes common to the corresponding conversations. A reverse mapping is possible by replacing distinct objects named in the action diagram by processes. An action is replaced by the corresponding conversation, with the set of processes in the conversation determined by the set of objects named in all the incoming and outgoing arcs of the action.

In order to support our hypothesis, we will discuss the way in which three major properties of a fault tolerant computation, namely, (1) freedom from interference, (2) backward recovery capability, and (3) crash resistance, are embodied in the OA and PM models.

(1) *Freedom from interference.* In the OA model, this requirement is ensured by the serializability property of actions and enforced by some concurrency control technique, such as two phase locking. In the PM model, freedom from interference between multiprocess computations structured as conversations is ensured by the two conversation rules, (i) a process can only communicate with those processes that are in the same conversation; and (ii) a process can only be inside a single conversation at a time (this rule can be relaxed under certain conditions, see later). The two phase locking discipline for

actions corresponds to entering a conversation (growing phase) and leaving a conversation (shrinking phase).

(2) *Backward recovery capability.* An action in progress can be aborted (recovered) without affecting any other ongoing actions. This recovery property of an action is enforced in conjunction with the concurrency control technique in use. In the case of two phase locking, this means that all the held locks are released simultaneously. This corresponds to the synchronized (simultaneous) exit from a conversation which is required from all the participating processes. The act of taking checkpoints at the start of a conversation has its dual in the OA model, and consists of the requirement of maintaining recovery data for objects used within an action. It was indicated earlier that the serializability property of actions can be maintained even if - for two phase locking - locks are released gradually (rather than simultaneously) during the shrinking phase of locking; however this has the danger of *cascade aborts* (recovery of an action can cause some other actions to be aborted as well). A similar observation can be made for conversations: the synchronized exit requirement is necessary to prevent cascade aborts. Fig. 8 illustrates that if

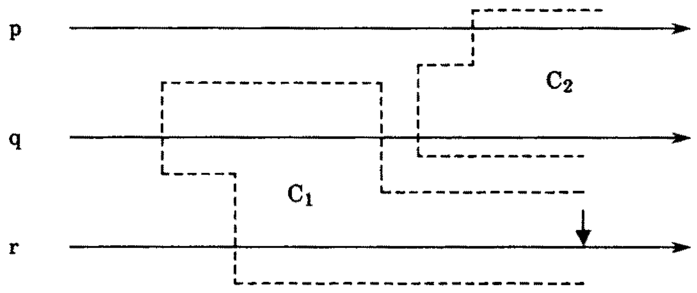

Fig. 8. Cascade aborts.

"conversations" C_1 and C_2 do not observe the rule of synchronized exit, and if time has advanced up to the point shown by the vertical arrow, and C_1 is to be aborted, then C_2 will have to be aborted as well.

(3) *Crash resistance.* A two phase commit protocol is employed in action based systems to ensure that despite the presence of failures such as node crashes, an action terminates either normally, with all the updated objects made stable to their new states, or abnormally with no state changes. A similar protocol will be required to ensure that the states of all the processes participating in a conversation are made stable.

A striking benefit of establishing the duality is that the body of knowledge and techniques developed for one model can be mapped and applied to the other model. We illustrate this with the help of the following two examples.

(1) *Read only requests.* A number of optimizations are possible if an action uses some or all of its objects in read only mode. Read locks can be released during the shrinking phase and need not be held till the end of the action, without the danger of cascade aborts. Further, no recovery data need be maintained for read only objects and they need not be involved in the two phase commit protocol since they do not change state. Such optimization strategies have been studied extensively within the context of database systems, e.g. [23]. However, to our knowledge, no such strategies have been studied for conversations, although they can be developed quite easily. Essentially, processes inside a conversation that do not update their states need not synchronize their exit from the conversation, nor do they need to take checkpoints at the start of the conversation. Consider a simple example. An action performs the following computation: $x := y + z$. Here y and z will be read locked; the commit protocol will involve only making object x stable to its new state and the action need generate no recovery data for y and z. Fig. 9 shows a

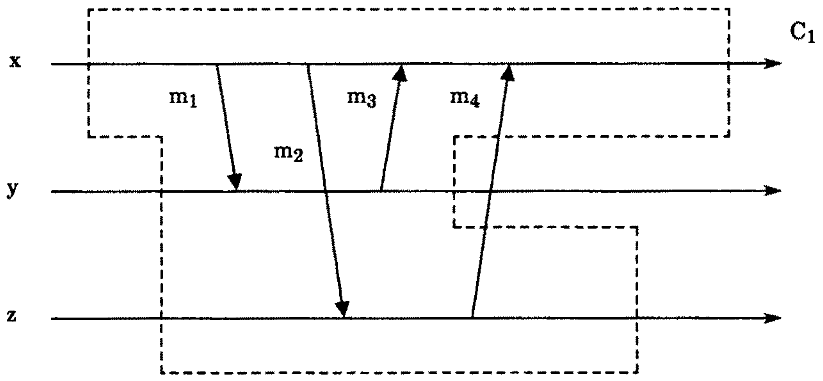

Fig. 9. Read only requests.

possible conversation to perform the same computation. In this particular case it is only necessary for process x to establish a checkpoint. Message m_1 (m_2) is a request to y (z) for some value, and message m_3 (m_4) contains the value sent by y (z).

Note that even though there is a two way exchange of messages between x and y (z), x can recover without affecting y (z), since message m_1 (m_2) is a *read* request. Indeed, y and z can take part in other conversations, while still in C_1, provided those conversations also involve only read requests directed to y and z. This is obviously the dual of the *shared read* lock mode rule applicable in the OA model. It is worth noting that, just as locking can cause deadlocks among actions, similar problems can occur in conversations.

(2) *Programmed exception handling.* So far we have examined the duality from the point of view of backward error recovery, which involves abandoning the current state for a prior state. In contrast, *forward error recovery* involves selective corrections to the current state to obtain an

acceptable state [21]. Programmed exception handling is a means of incorporating this form of forward recovery. A widely accepted exception handling strategy is as follows: if during the execution of a computation an error is detected (an exception is detected) for which a specifically programmed *handler* is available, then that handler is invoked; if there is no programmer-provided handler available then a *default handler* is invoked whose function is to invoke backward recovery. Thus, exception handling can provide a uniform means of incorporating both forward and backward error recovery strategies [24,25]. A recent paper [26] proposes an exception handling strategy for concurrent processes with conversations and describes how processes can resolve concurrent exceptions through the use of exception trees. To keep this paper brief, we will not describe this strategy; instead we note here that these exception handling ideas, although developed using the PM model, have since been applied by Taylor [12] to the OA model.

A summary of the various characteristics of the two models for which duality has been established is presented in Table 1.

Object-Action Model	*Process-Message Model*
Objects	Processes
Actions	Conversations
RPCs	send-receive messages
concurrency control for serializability	conversation rules preventing no outside communication
stable objects	stable processes
growing phase (2-phase locking)	processes entering a conversation
shrinking phase (2-phase locking)	processes leaving a conversation
read locks	read only request messages

Table 1. Duality Mapping.

5. Examples

This section contains two further examples, one taken from the database area and normally programmed using objects and actions and the other taken from the process control area and normally programmed using processes and messages. It will be shown that programs written using the primitives of one model have duals in the other. Simple and self-explanatory notation will be used for program description.

Banking application. An example often used to illustrate the properties of an action concerns transferring a sum of money from one bank account to another. The failure atomicity property, for example, will ensure that either the sum of money is debited from one account and credited to the other, or no state changes are produced. For the sake of illustration, the application has been structured to invoke nested actions, even though simpler, non-nested solutions are clearly possible.

Two types of objects will be assumed: standing-order, and credit-debit:

```
type standing-order = object
            - - object variables - -
  action transfer (to, from: credit-debit; amount: dollars)
            cobegin
            authority (to, from);
            to.credit (amount);
            from.debit (amount)
            coend
  end action
            - - other actions, e.g. authority - -
end standing-order;


type credit-debit = object
            - - current account variables - -
  action credit (amount:dollars)
            - - add amount - -
  end action

  action debit (amount:dollars)
            - - subtract amount - -
  end action
            - - other actions - -
end credit-debit;
```

Specific instances of these objects can be created:

```
            order : standing-order;
            accl, acc2 : credit-debit;
```

An invocation of *order.transfer* will give rise to a nested computation shown in Fig. 10. Any exceptions during the execution of transfer will cause that action to be aborted.

The same program can be recoded quite easily in terms of communicating processes.

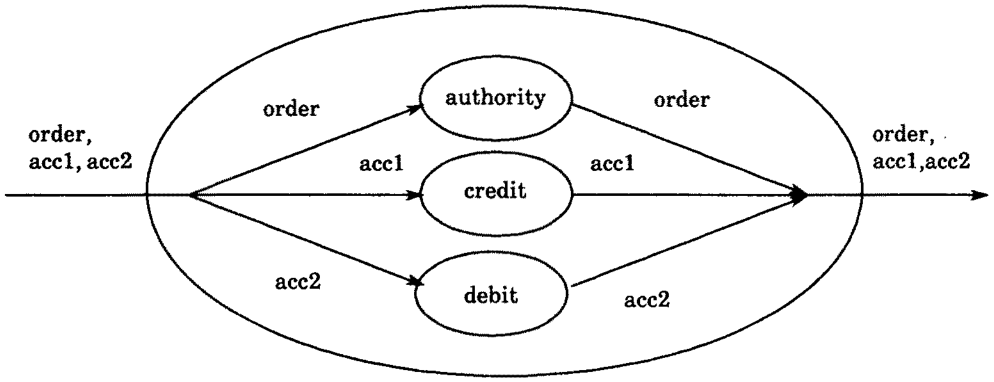

Fig. 10. A bank action.

```
type standing-order = process
            - - process variables - -
select
  conversation transfer (to, from: credit-debit; amount: dollars)
            cobegin
            send (self, authority, to, from);
            send (to, credit, amount);
            send (from, debit, amount)
            coend
  end conversation
- - other selections, e.g. authority - -
end select
end standing-order

type credit-debit = process
            - - current account variables - -
select
  conversation credit (amount: dollars)
        - - add amount - -
  end conversation
- - other selections, e.g. debit - -
end select
end credit-debit
```

Specific instances of these processes can be created:

```
order : standing-order;
acc1, acc2 : credit-debit;
```

A transfer conversation can be initiated by sending a message to order:

```
send(order,transfer,parameters)
```

The transfer conversation is shown in Fig. 11.

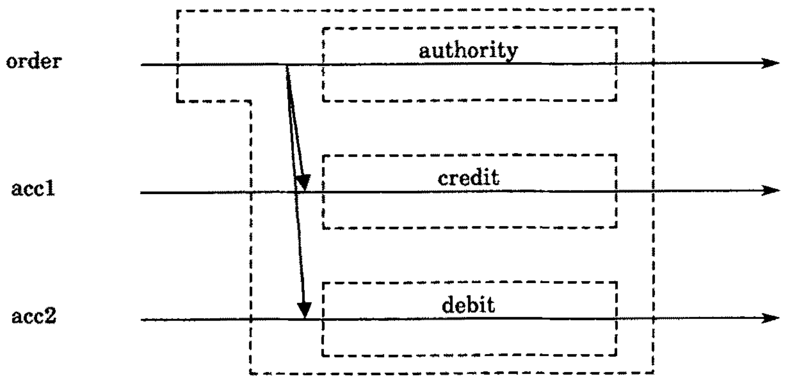

Fig. 11 . A bank conversation.

The second example is taken from a process control application in the coal mining industry [27]. Fig. 12 shows a simplified pump installation. It is used to pump mine-water collected in the

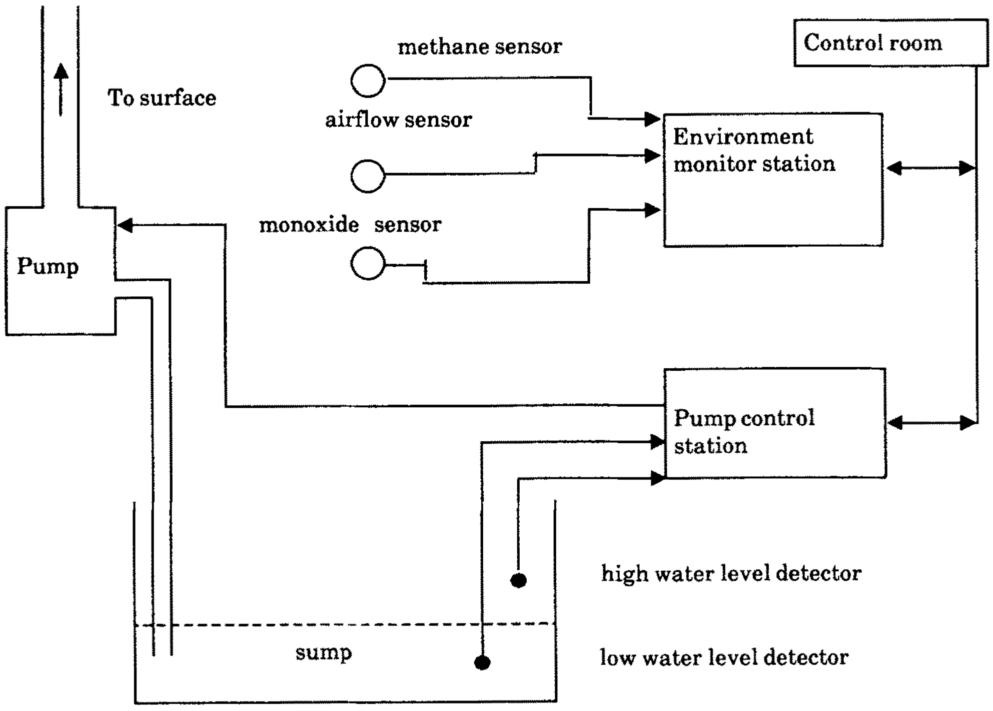

Fig 12. Pump Control System.

sump at the shaft bottom to the surface. The pump is enabled by a command from the control room. Once enabled, it works automatically, controlled by water level sensors; detection of a high level causes the pump to run until a low level is indicated. For safety reasons, the pump must not run if the percentage of methane exceeds a certain safety limit. Some other parameters of the environment are also monitored by the monitoring station.

The control software can be structured as five communicating processes, namely: pump controller, surface, level, pump and monitor. Some sketchy details are given here for the pump controller.

Pump controller process. Some of its functions are to receive start/stop command from the *surface* process (representing the control room), receive water level reports from the *level* process and to receive an alarm signal from the *monitor* process. The *controller* process can send start/stop commands to the *pump* process which controls the pump.

A study of process structure discussed in [27] reveals that the overall behaviour of the other processes have a similar structure to the pump-controller, either receiving requests to carry out certain functions and/or sending messages to other processes to request certain functions to be performed. These interactions can be organized as conversations. A highly simplified program fragment for the pump controller is given in Fig. 13.a.

(a) Process-Message Model	*(b) Object-Action Model*
type *pump-controller* = **process**	**type** *pump-controller* = **object**
- - *process variables* - -	
select	- - *object variables* - -
conversation on/off (...)	**action** on/off (...)
send start/stop command	*send start/stop command*
to the pump process	*to the pump process*
end conversation	**end action**
- - - other selections - - -	- - - other actions - - -
end select	
end *pump-controller*	
	end *pump-controller*

Fig. 13. Pump-controller example.

A command to enable or disable the pump from the surface process starts a conversation containing the pump-controller and the pump process: if the conversation terminates normally, the pump will have changed state accordingly. It is fairly easy to reprogram this example in terms of objects and actions, with the five processes replaced by the corresponding objects. For the sake of illustration, the program for the pump-controller object is shown in Fig. 13.b.

These examples provide further empirical support to our claim by illustrating that close similarity exists between the two classes of programs. Given a program constructed from the primitives defined by one model, it can be mapped directly into a dual program of the second model.

6. Concluding Remarks

After examining the structure of a variety of systems, two canonical models of fault tolerant systems were developed, one of which is representative of the techniques and terminology used within the database and office information systems community, the other of which is more closely allied to the real time and process control applications area. These models were shown to be duals of each other. Although, in retrospect, this may not appear to be a surprising conclusion, particularly given the Lauer and Needham paper, we had not before realized how direct and complete the relationship between the two models was, and are not aware of any earlier literature explaining and exploiting this duality. Instead, one finds that fault tolerant systems are constructed and described using the concepts and terminology applicable to just one of the two models, with no apparent realization of the potential relevance of systems and the literature describing them which make use of the other model. However, we must admit that the duality that we have discussed is sometimes obscured by the fact that many process control applications are structured as a small and fixed number of processes, whereas it is more usual to find object based systems which contain a large and dynamically varying number of objects.

Our arguments to support the duality claim were based on an examination of three properties of a fault tolerant computation, namely: freedom from interference, backward recovery capability and crash resistance. It was shown that mechanisms employed to implement a given property in one model have duals in the other. Similarly, any particular behaviour observed in one model has its dual in the other. Examples presented in the paper show that programs developed using the primitives of one model can be mapped easily to the programs of the other model. Indeed, we would claim that the differences between the two models are principally a matter of view point and terminology.

The establishment of the equivalence between the two approaches to fault tolerance has several interesting implications, some of which are enumerated here.

(1) There seems to be no inherent reason for favouring one approach over the other. For example, there is no obvious reason why a real time system must be designed using the primitives of the PM model. In fact, one is led to state that the choice of a model to adopt for a given system should not be dictated by the application area but by the architectural features of the layer over which the system is to be built.

(2) It can also be stated that a single system architecture based on either model can in principle, support both classes of applications.

(3) We further speculate that, were sufficient representative systems of each class available for detailed evaluation and comparison, we would find that the observation made in [2] regarding the invariance of operating system performance under two classes of systems also applies to this fault tolerance duality.

(4) Techniques and mechanisms which happen to have been developed within the domain of just one of the models can be mapped and applied to the other model. Two examples were presented to illustrate this observation. It was shown that optimization techniques developed for read operations of actions can be applied to optimize conversations. A second example indicated that the exception handling framework developed for the PM model can be applied to the OA model.

(5) We put forward another proposal for further investigation. There is a large body of literature on the topic of replicated object management for increasing availability. We believe that interesting techniques for replicated process management can be developed from these studies and applied to process control systems that have been developed using the PM model.

(6) The ideas from this paper can be used for the design of fault tolerant systems with minimum set of compatible concepts, thus allowing several degrees of freedom in the design process to be eliminated, leading to well structured systems.

(7) Finally, given that, as discussed in [28], there is the prospect of using certain kinds of fault tolerance techniques to provide increased security and not just increased reliability, it appears that the duality mapping presented here can be extended and applied to clarify and illuminate at least some of the literature discussing various approaches to building multi-level secure systems. This however is a topic which will not be explored further in this paper.

Acknowledgements.

The authors have had discussions and arguments with several of their colleagues over a period of many years on the subject matter reported here. Those we would like to mention specifically include Graham Wood, David Taylor and Roy Campbell. Written comments from Tom Anderson and Robert Stroud on an early draft of the paper are also gratefully acknowledged. This work was supported in part by research grants from SERC/Alvey and the UK Ministry of Defence.

References

[1] J.N. Gray, "An approach to decentralized computer systems", IEEE Trans. on Soft. Eng., SE-12, No.6, 1986, pp.684-689.

[2] H.C. Lauer and R.M. Needham, "On the duality of operating system structures", Proc. of 2nd Int. Symp. on Operating Systems, INRIA, Oct. 1978; reprinted in ACM Operating System Review, Vol. 13, April 1979, pp. 3-19.

[3] B. Liskov and R. Scheifler, "Guardians and actions: linguistic support for robust distributed programs", ACM TOPLAS, Vol. 5, No. 3, 1983, pp.381-404.

[4] A.Z. Spector et al, "Support for distributed transactions in the TABS prototype", IEEE Trans. on Soft. Eng., SE-11, No. 6, 1985, pp.520-530.

[5] L. Svobodova, "Resilient distributed computing", IEEE Trans. on Soft. Eng., SE-10, No.3, 1984, pp.257-268.

[6] S.K. Shrivastava, "Structuring distributed systems for recoverability and crash resistance", IEEE Trans. on Soft. Eng., SE-7, No. 4, 1981, pp.436-447.

[7] K.P. Birman, "Replication and fault tolerance in the ISIS system", Proc. of 10th Symp. on Princ. of Op. Sys., ACM Operating Systems Review, 19, No. 4, 1985, pp.79-86.

[8] E. Nett et al, "Profemo: design and implementation of a fault tolerant distributed system architecture", GMD Studien, No. 100, Tech. report, GMD, St. Augustine, 1985.

[9] K. Eswaren et al, "On the notions of consistency and predicate locks in a database system", CACM, 19, No. 11, 1976, pp.624-633.

[10] E. Best and B. Randell, "A formal model of atomicity in asynchronous systems", Acta Informatica, 16, 1981, pp.93-124.

[11] C.T Davies, "Recovery semantics for a DB/DC system", Proc. of ACM Nat. Conf., 1973, pp. 136-141.

[12] D.J. Taylor, "Concurrency and forward recovery in atomic actions", IEEE Trans. on Soft. Eng., SE-12, No. 1, 1986, pp.69-78.

[13] C.A.R. Hoare, "Communicating sequential processes", CACM, 21, No. 8, 1978, pp.666-677.

[14] D.L. Russell, "State restoration in systems of communicating processes", IEEE Trans. on Soft. Eng., SE-6, No. 2, 1980, pp.183-193.

[15] K.H. Kim, "Approaches to mechanization of the conversation scheme based on monitors", IEEE Trans. on Soft. Eng., SE-8, No. 3, 1982, pp.189-197.

[16] S.K. Shrivastava and J.P. Banatre, "Reliable resource allocation between unreliable processes", IEEE Trans. on Soft. Eng., SE-4, No.3, 1978, pp.230-241.

[17] W.G. Wood, "A decentralized recovery control protocol", Digest of papers, FTCS-11, Portland, 1981, pp.159-164.

[18] R. Koo and S. Toueg, "Checkpointing and rollback recovery for distributed systems", IEEE Trans. on Soft. Eng., SE-13, No.1, 1987, pp.23-31.

[19] B. Randell, "System structure for software fault tolerance", IEEE Trans. on Soft. Eng., SE-1, No.2, 1975, pp.220-232.

[20] K.M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems", ACM TOCS, 3, No.1, 1985, pp.63-75.

[21] B. Randell, P.A. Lee and P.C. Treleaven, "Reliability issues in computing system design", ACM Comp. Surveys, 10, No. 2, 1978, pp.123-166.

[22] T. Anderson and J.C. Knight, "A framework for software fault tolerance in real time systems", IEEE Trans. on Soft. Eng., SE-9, No. 3, 1983, pp.355-364.

[23] C. Mohan and B.G. Lindsay, "Efficient commit protocols for the tree of processes model of distributed transactions", Proc. of 2nd ACM Symp. on Princ. of Dist. Comp., Montreal, 1983, pp.76-88.

[24] F. Cristian, "Exception handling and software fault tolerance", IEEE Trans on Computers, C-31, No. 6, 1982, pp.531-540.

[25] T. Anderson and P.A. Lee, "Fault Tolerance: Principles and Practice", Prentice Hall, London, 1981.

[26] R.H. Campbell and B. Randell, "Error recovery in asynchronous systems", IEEE Trans. on Soft. Eng., SE-12, No.8, 1986, pp.811-826.

[27] M. Sloman and J. Kramer, "Distributed systems and computer networks", Prentice Hall, London, 1987.

[28] J.E. Dobson and B. Randell, "Building reliable secure computing systems out of unreliable insecure components", Proc. of IEEE Symp. on Security and Privacy, Oakland, CA, April 1986, pp.187-193.